JEFFERSON PAULO KOPPE

HYPERDHT - DHT DE UM SALTO BASEADA EM HIPERCUBO VIRTUAL DISTRIBUÍDO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Luis Carlos E. de Bona
Co-Orientador: Prof. Dr. Elias P. Duarte Jr.

CURITIBA

2013

SUMÁRIO

LISTA DE FIGURAS **iv**

LISTA DE TABELAS **v**

RESUMO **vi**

ABSTRACT **vii**

1 INTRODUÇÃO **1**

2 TABELAS HASH DISTRIBUÍDAS **5**

2.1 Visão Geral das Redes P2P 5

2.2 Redes P2P Baseadas em Tabelas Hash Distribuídas 7

2.3 Tabelas Hash Distribuídas de Múltiplos Saltos 12

2.3.1 CAN 13

2.3.2 Chord 15

2.3.3 Tapestry 18

2.4 Tabelas Hash Distribuídas de Salto Único 19

2.4.1 OneHop DHT 20

2.4.2 D1HT 21

3 ALGORITMOS DE DIAGNÓSTICO DISTRIBUÍDO **24**

3.1 Definições Preliminares 24

3.2 Diagnóstico Adaptativo e Distribuído 26

3.3 Algoritmos Hierárquicos de Diagnóstico Distribuído 27

3.3.1 O Algoritmo DiVHA e a Rede HyperBone 30

4 HYPERDHT: DHT DE UM SALTO BASEADA EM HIPERCUBO VIRTUAL DISTRIBUÍDO 34

4.1 Considerações Preliminares 34

4.2 Modelo do Sistema 37

4.3 Particionamento do Espaço de Chaves *Hash* 39

4.4 Consultas em um Único Salto 44

4.5 Esquema de Replicação 46

4.6 Protocolo de Entrada em uma Rede HyperDHT 47

4.7 Saída de Participantes do HyperDHT 51

5 RESULTADOS EXPERIMENTAIS 53

5.1 Primeira Série de Simulações 53

5.1.1 Latência e Tempo para Disseminação de Eventos 54

5.1.2 Consumo dos Recursos de Rede 56

5.2 Segunda Série de Simulações 59

5.2.1 Análise do *Churn* na Taxa de Sucesso das Consultas 60

5.2.2 Intervalo de Testes e o Consumo dos Recursos de Rede 61

6 CONCLUSÕES 63

REFERÊNCIAS BIBLIOGRÁFICAS 68

LISTA DE FIGURAS

1.1 Rede virtual HyperDHT de 8 nodos. 3

2.1 Exemplo de uma rede sobreposta com topologia baseada em um hipercubo. 6

2.2 Geração das chaves (IDs) e armazenagem de objetos em uma DHT. 9

2.3 Resultado da função criptográfica e efeito avalanche da função criptográfica SHA-1. 9

2.4 Exemplo do particionamento do espaço de chaves utilizado pelo Chord. . . 10

2.5 Exemplo de roteamento de uma consulta em uma DHT estruturada. 11

2.6 Exemplo de roteamento do nodo X para o nodo E em um sistema CAN. . . 13

2.7 Exemplo da entrada do nodo Z em um sistema CAN de 2 dimensões. . . . 14

2.8 Representação de um sistema Chord com 10 participantes e 5 objetos. . . . 16

2.9 Exemplo de uma Finger Table para *peer* 8. 17

2.10 Sequência de resolução dígito por dígito do Tapestry. 18

2.11 Exemplo do caminho de roteamento de uma mensagem em uma DHT Tapestry [37]. 19

2.12 Fluxo das notificações de eventos em um sistema OneHop [1]. 21

2.13 Disseminação de eventos em um sistema D1HT [23]. 23

3.1 Um exemplo de assinalamento de testes; o nodo 2 está falho. 25

3.2 Exemplo de uma rodada de testes realizada pelo algoritmo Adaptive-DSD. 27

3.3 Sistema de 8 nodos agrupados em clusters. 28

3.4 Grafo $H(S)$ para um sistema de 8 *peers* não-falhos. 30

3.5 Grafo $T(S)$ gerado pelo DiVHA; linha pontilhada indica uma conexão adicional. 31

3.6 Algoritmo de assinalamento de testes: DiVHA. 32

4.1 Grafo $T(S)$ para um sistema de 8 vértices *ocupados*. 38

4.2 Representação gráfica do particionamento do espaço de chaves em um sistema HyperDHT. 39

4.3 Identificadores binários e cálculo da distância entre vértices em um hipercubo. 40

4.4 Pseudo-código - *peer* responsável pelas chaves associadas a um vértice. . . . 41

4.5 Divisão do espaço de chaves na entrada de um novo *peer*. 42

4.6 Exemplo de particionamento do espaço de chaves em um sistema HyperDHT. 43

4.7 Pseudo-código - algoritmo *lookup* do HyperDHT. 45

4.8 Pseudo-código - replicação de valores em um sistema HyperDHT. 47

4.9 Uma rede HyperDHT recém criada e inicializada. 48

4.10 Procedimento de entrada de um novo *peer* em uma rede HyperDHT. 48

4.11 Pseudo-código - algoritmo de incremento do sistema em uma dimensão. . . 49

4.12 Expansão de um sistema HyperDHT. 49

5.1 Latência média com intervalo de confiança de 95%. 54

5.2 Latência média em segundos, para intervalo de testes de 30s. 56

5.3 Uso total dos recursos de rede para manutenção do HyperDHT. 57

5.4 Uso médio dos recursos de rede por peer. 58

5.5 Impacto do *Churn* na taxa de sucesso de *gets* em salto único. 60

5.6 Análise do intervalo de testes no consumo da banda de rede. 62

LISTA DE TABELAS

2.1 Mapa de vizinhos mantido pelo nodo 67493 [37]. 18

3.1 Resultado da função $C(i, s)$ para um sistema de 8 nodos. 28

4.1 Exemplo 1 - Resultados da função find_vertice_owner. 42

4.2 Exemplo 2 - Resultados da função find_vertice_owner. 42

RESUMO

Um problema chave das redes P2P é a localização de um *peer*, ou nodo, que armazena um determinado recurso ou conteúdo. Os primeiros sistemas P2P desenvolvidos utilizavam métodos de localização por inundação ou passeios aleatórios, que não garantem que a informação procurada será encontrada, mesmo que ela exista na rede. As DHTs (*Distributed Hash Table*) são redes P2P que oferecem uma solução mais eficiente e escalável para localização de informações, através de um serviço de *lookup* similar ao encontrado em tabelas *hash*, onde os pares (chave, valor) são armazenados de forma distribuída. O tempo necessário para a localização de um *peer* em um sistema DHT é mensurado em saltos e corresponde à quantidade de *peers* consultados até que a resposta seja obtida. Este trabalho propõe uma nova abordagem para DHTs de salto único, batizada de HyperDHT. O HyperDHT é baseado em um hipercubo virtual distribuído, formado por nodos espalhados pela Internet interligados por enlaces virtuais. O HyperDHT utiliza a rede de sobreposição construída pelo algoritmo DiVHA (*Distributed Virtual Hypercube Algorithm*), que é um algoritmo de diagnóstico distribuído hierárquico com limites bem conhecidos e definidos para o número de testes executados e para a latência que, no pior caso, é de $log_2 N$ rodadas de testes. Diferentemente do que é usualmente empregado nos sistemas DHTs, o posicionamento de um novo participante no HyperDHT é realizado de forma determinística, a fim de posicioná-lo no local da rede onde ele é mais necessário. A rede sobreposta forma a base na qual é realizado o particionamento, o balanceamento e o mapeamento consistente das chaves *hash*. Trata-se de uma rede dinâmica que permite a entrada e saída de participantes, com mecanismos correspondentes para posicionamento e busca dos objetos. O HyperDHT foi implementado em um ambiente de simulação e resultados experimentais são apresentados para latência de disseminação de eventos, a sobrecarga em termos de recursos consumidos na rede e a análise do *churn* na taxa de sucesso das consultas.

ABSTRACT

A key problem of P2P networks is the location of a peer (or node), which stores a particular resource or content. Early P2P systems employed location methods based on flooding or random walks, which do not guarantee that the information sought is found, even if it is somewhere in the network. A Distributed Hash Table (DHT) is a P2P network that provides a more efficient and scalable solution for finding information, through a lookup service similar to that of hash tables, but the (key, value) pairs are distributed across the network. The time required for the location of information in a DHT system is measured in hops and represents the amount of peers will be consulted until the response is obtained. This dissertation proposes a new single hop DHT, named HyperDHT. The HyperDHT is based on a distributed virtual hypercube, consisting of nodes spread across the Internet connected by virtual links. The HyperDHT uses the overlay network constructed by the Distributed Virtual Hypercube Algorithm (DiVHA), which is a hierarchical distributed diagnosis algorithm which presents several logarithmic properties. HyperDHT is different from most other strategies in that the position of new participants is deterministic, a new participant is located where it is most needed. The overlay network allows the hash keys to be partitioned, balanced and consistently mapped. The network is dynamic allowing participants to enter and leave in the system, at the same time managing object position and deploying an efficient search strategy. HyperDHT was implemented in a simulation environment and experimental results are presented for the latency of event dissemination, the overhead in terms of network resources consumed, and the search success rate in the presence of churn.

CAPÍTULO 1

INTRODUÇÃO

As redes *Peer-to-Peer* (P2P) surgiram originalmente como uma alternativa ao tradicional modelo cliente/servidor [2] para o compartilhamento em larga escala de recursos e conteúdo em redes, impulsionado pela popularização da Internet. O modelo P2P puro refere-se a um sistema totalmente distribuído, no qual todos os participantes possuem a mesma funcionalidade, operando tanto como cliente quanto como servidor [21]. Esta arquitetura provê mecanismos para localização de informações em sistemas distribuídos, sem requerer a intermediação ou suporte de uma entidade centralizadora. Em geral, as redes P2P são projetadas para funcionar mesmo que uma grande quantidade de nodos entrem e saiam do sistema a todo momento.

Os primeiros sistemas P2P desenvolvidos utilizavam métodos de localização de informação baseados em técnicas de inundação (p.ex. Gnutella [25]), passeios aleatórios e outros métodos similares [8], os quais não são escaláveis ou não garantem que a informação procurada será encontrada, mesmo que ela exista no sistema [2]. Os sistemas DHT (*Distributed Hash Table*) foram propostos como uma melhoria para esta abordagem, pois oferecem uma solução mais eficiente e escalável para localização de informações [12].

As DHTs disponibilizam uma função de consulta similar à encontrada em tabelas *hash* tradicionais, sendo que as chaves são distribuídas pelos diversos nodos do sistema. Para encaminhar uma consulta a partir do nodo requerente até aquele responsável pela chave procurada (nodo destino), as DHTs fazem uso de tabelas de roteamento. Cada vez que uma consulta é encaminhada para um novo nodo é dito que houve um salto (ou do inglês *hop*). De maneira geral, quanto maior a quantidade de saltos necessários para resolver uma consulta, maior será a latência, ou seja, maior será o intervalo de tempo para que o

nodo que originou a consulta obtenha a sua resposta.

O uso de grandes tabelas de roteamento permite que as consultas sejam resolvidas em um número menor de saltos, mas se deve considerar que quanto maior a quantidade de informação de roteamento armazenada em cada nodo, maior será a demanda de comunicação para sua manutenção [16]. Deste modo, as DHTs devem balancear a latência das consultas com a demanda de comunicação para manutenção das tabelas de roteamento.

As primeiras DHTs, a exemplo do Pastry [33], Tapestry [37], Chord [15] e CAN [31], são classificadas como *Multi Hop DHT* pois fazem uso de tabelas de roteamento parciais e necessitam que uma consulta seja roteada por diversos nodos. As DHTs cujas tabelas de roteamento contêm informações sobre todos os participantes do sistema são conhecidas por *Single Hop DHT*, e o que as diferencia das multi-hop é a capacidade de resolver as consultas em um único salto, usando apenas a tabela de roteamento local.

Uma *Single Hop DHT* requer que os eventos de entrada, saída ou falha de qualquer nodo do sistema sejam reportados para todos os participantes do sistema [1]. Por este motivo, a preocupação principal passa a ser como realizar a disseminação dos eventos de maneira rápida e eficiente [16]. As DHTs de salto único têm como principal compromisso minimizar ao máximo a latência das consultas, mas não devem comprometer o sistema com o consumo excessivo de recursos de rede para a manutenção de suas tabelas de roteamento.

O sistema OneHop [1] foi a primeira DHT proposta a garantir que a maior parte das consultas sejam resolvidas com um único salto. A hierarquia empregada pelo OneHop minimiza os custos de manutenção das tabelas de roteamento, mas causa um alto nível de desbalanceamento na carga de atividades exercidas pelos diversos nodos. O sistema D1HT [23] é uma proposta mais recente que, diferentemente da OneHop, não atribui funções distintas entre os nodos. A rede de sobreposição construída pelo D1HT é similar à empregada pelo Chord, e um esquema parecido com a *Finger Table* utilizada no Chord permite a propagação de eventos em tempo logarítmico.

Uma característica comum entre as DHTs de salto único propostas até o momento,

é o fato de existirem situações que impossibilitam uma consulta ser respondida em um único salto, requerendo saltos adicionais para que a resposta seja obtida [29]. Em geral, esta situação é ocasionada quando um ou mais eventos estão em fase de disseminação na rede, período este no qual os nodos diferentes podem ter visões ligeiramente divergentes da configuração do sistema.

Este trabalho apresenta uma nova DHT de salto único, chamada de HyperDHT. A proposta do HyperDHT é prover uma solução onde o procedimento de localização de uma informação ou um objeto distribuído em uma rede P2P seja realizado de forma rápida, eficiente e escalável. Sendo o HyperDHT uma DHT de salto único, as questões relativas à escalabilidade não estão relacionadas com o algoritmo de localização, mas sim voltadas aos procedimentos realizados para manutenção da tabela de roteamento.

Algumas técnicas empregadas por algoritmos hierárquicos de diagnóstico distribuído são utilizadas pelo HyperDHT para a construção da rede sobreposta e para a manutenção das tabelas de roteamento. Os participantes são dispostos em uma estrutura modelada em um hipercubo virtual, conforme ilustra a figura 1.1.

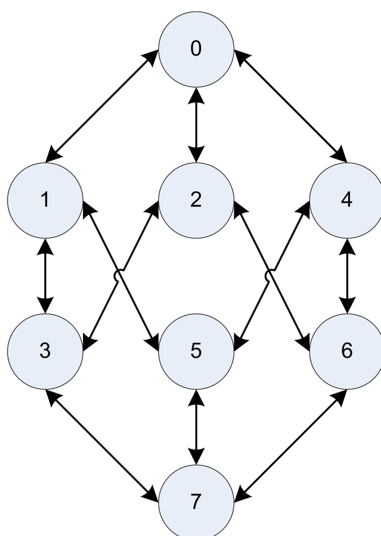

Figura 1.1: Rede virtual HyperDHT de 8 nodos.

A estrutura que forma a rede sobreposta tem características que permitem implementar um sistema onde os nodos sem falha conseguem determinar o estado de todos os participantes em um tempo máximo de log_2N rodadas de testes e, o número máximo de

testes por rodada é de $N \times log_2 N$. Os limites máximos de latência e quantidade de testes são formalmente provados em [22].

A garantia da latência máxima proveniente do sistema de diagnóstico, somado ao fato de todos os nodos terem conhecimento completo do sistema, habilitam o HyperDHT a adotar estratégias para maximizar a chance das consultas serem respondidas em um único salto, mesmo nas situações em que o sistema não está estabilizado, com eventos ainda em fase de disseminação.

O HyperDHT incorpora no protocolo de entrada mecanismos para posicionar deterministicamente um novo participante no local da rede onde ele é mais necessário. Esta abordagem diferencia o HyperDHT das demais DHTs que, usualmente, deixam o problema do posicionamento de um novo participante por conta da função *hash*, acreditando que o efeito avalanche irá fazer uma distribuição razoavelmente homogênea dos participantes na rede de sobreposição. Como resultado, podemos esperar que o balanceamento de carga em um sistema HyperDHT seja mais homogêneo, comparado com as DHTs que utilizam o *hash* para posicionar os participantes na rede.

O restante deste trabalho está organizado da seguinte maneira: o capítulo 2 introduz as DHTs e algumas das principais abordagens relacionadas às DHTs existentes; o capítulo 3 apresenta os algoritmos de diagnóstico distribuído, em especial o algoritmo DiVHA, o qual está diretamente relacionado a este trabalho; no capítulo 4 temos a apresentação e a especificação do HyperDHT; o capítulo 5 traz uma análise dos resultados das simulações e, por fim, a conclusão e trabalhos futuros são expostos no capítulo 6.

CAPÍTULO 2

TABELAS HASH DISTRIBUÍDAS

O presente capítulo é iniciado com uma breve introdução sobre redes P2P, seguida pela apresentação dos conceitos básicos das DHTs. Na sequência são apresentados alguns exemplos de DHTs de múltiplos saltos e o capitulo é finalizado com uma discussão sobre as DHTs de salto único.

2.1 Visão Geral das Redes P2P

Os sistemas par-a-par, do inglês *peer-to-peer* (P2P), surgiram como uma alternativa ao tradicional modelo cliente/servidor [2]. O modelo P2P puro refere-se a um sistema totalmente distribuído, no qual todos os participantes do sistema possuem o mesmo papel, operando tanto como cliente quanto como servidor. Esta arquitetura é projetada para o compartilhamento de recursos computacionais por troca direta e não requer a intermediação ou suporte de uma entidade centralizadora. Um recurso computacional, como um objeto a ser compartilhado, pode ser um arquivo, uma informação, um dado de uma base de dados ou, até mesmo, ciclos de CPU.

Os participantes de um sistema P2P podem se conectar e desconectar de forma dinâmica, fenômeno denominado *churn* [32]. O *churn* tem impacto direto na solução adotada para localização dos objetos compartilhados dentro de um sistema P2P, pois a entrada e saída constante dos participantes faz com que os objetos mudem de localização, tornando a busca destes objetos um desafio.

A arquitetura dos sistemas P2P é baseada nas redes de sobreposição *(overlay networks)*, nas quais os canais de comunicação de uma rede física existente são usados

para criar uma topologia lógica virtual. A topologia lógica das redes de sobreposição descreve como os participantes devem se conectar e como o tráfego das mensagens trocadas entre eles deve fluir. Os participantes, também chamados de nodos, são os processos que compõem a rede de sobreposição. O intuito é criar uma estrutura de nível mais elevado de abstração, que permita uma solução mais simples e adequada de determinados problemas [6] como, por exemplo, a localização de recursos computacionais distribuídos dinamicamente em uma rede de computadores.

A figura 2.1 ilustra uma rede sobreposta de 8 nodos estruturada em um hipercubo sobre a camada de uma rede física (p.ex., Internet). As estruturas na base da ilustração representam a rede física que é formada pelos roteadores e suas conexões. Na parte superior temos os participantes da rede sobreposta e as conexões lógicas entre eles. Observe que dois nodos vizinhos na rede sobreposta não são necessariamente vizinhos na rede física. Os nodos 6 e 7, por exemplo, que são vizinhos um do outro na rede sobreposta, não são vizinhos na rede física.

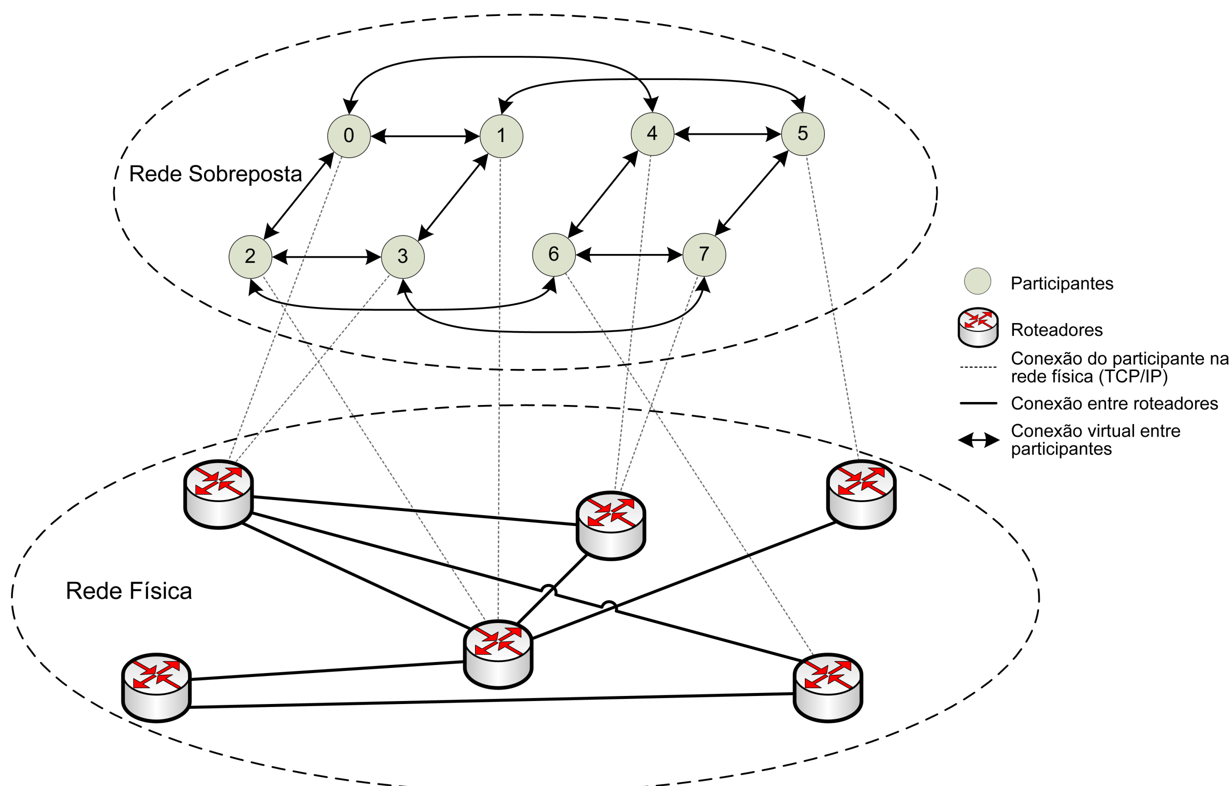

Figura 2.1: Exemplo de uma rede sobreposta com topologia baseada em um hipercubo.

Dependendo do mecanismo utilizado para construção da rede de sobreposição, um sistema P2P pode ser classificado como não-estruturado ou estruturado [2, 21]. Os sistemas P2P *não-estruturados* têm a topologia de rede virtual determinada de maneira *ad hoc* e os nodos estabelecem conexões com seus pares arbitrariamente, sendo que não existem regras pré-estabelecidas para o posicionamento dos nodos na topologia. Nestas redes, os mecanismos de consulta para a localização dos recursos compartilhados são baseados em técnicas de inundação (Gnutella [25]), passeios aleatórios ou outras técnicas [8]. Estas abordagens possuem limitações na escalabilidade e não fornecem garantia de que o objeto procurado será encontrado, mesmo que ele exista, pois não há a certeza de que uma consulta alcançará todos os participantes do sistema. No entanto, as redes de sobreposição não-estruturadas, por não possuírem uma topologia definida, têm custos de manutenção baixos se comparados com as redes estruturadas [2].

As redes P2P *estruturadas* têm a sua topologia da rede virtual formada através de regras pré-estabelecidas. A topologia destas redes pode ser implementada de diversas formas, como em malha, no Pastry [33] e Tapestry [37]; em anel, no Chord [15]; em *toro d-dimensional*, no CAN [31]; entre outros. Os principais protocolos desta categoria fazem uso das tabelas de indexação distribuídas.

2.2 Redes P2P Baseadas em Tabelas Hash Distribuídas

As DHTs *(Distributed Hash Tables)* [4] são estruturas de dados distribuídas que associam um identificador único para os dados, de modo a permitir sua indexação e localização determinística. As DHTs oferecem uma solução descentralizada e escalável para o problema da localização de objetos em sistemas amplamente distribuídos, além de garantir que toda consulta que tenha solução seja resolvida. As primeiras DHTs (CAN [31], Chord [15], Pastry [33] e Tapestry [37]) foram introduzidas quase simultaneamente em 2001. Desde então, DHTs têm sido uma área de pesquisa de grande interesse do meio acadêmico, cujos estudos e técnicas desenvolvidas vêm sendo adotados por aplicativos P2P, em especial aqueles destinados para o compartilhamento de arquivos.

As DHTs permitem a construção de sistemas distribuídos em conformidade com os requisitos básicos de uma rede P2P, como: (1) a descentralização, uma vez que não existe nenhum tipo de coordenação central nas DHTs; (2) a escalabilidade, que se dá em razão da capacidade de auto-organização das DHTs; e (3) a tolerância a falhas, visto que a integridade da rede sobreposta e dos serviços oferecidos pelo sistema DHT não são comprometidos pelo *churn*.

A construção de uma DHT se dá de forma similar a uma tabela *hash*, também conhecida por tabela de dispersão ou tabela de espalhamento. Uma tabela *hash* [34] é uma estrutura de dados especial que usa uma função *hash* para associar uma identificação, conhecida por chave, a valores. A função *hash* é empregada para transformar as chaves em índices de um vetor, no qual os valores são armazenados. O objetivo é, a partir de uma chave, fazer uma busca rápida para obter o valor desejado.

Nas DHTs a tabela *hash* é particionada para operar de forma distribuída em uma rede, onde tanto os participantes quanto os objetos recebem uma chave gerada pela função *hash*. A chave do participante define a posição que ele deverá ocupar na topologia da rede sobreposta e a porção da tabela *hash* pela qual ele é responsável. A chave associada ao objeto define de forma determinística o lugar na rede onde ele será alocado.

A figura 2.2 exemplifica este processo. Considere o dado 2 como sendo um arquivo qualquer, o nome deste arquivo (ou o seu conteúdo) é passado como parâmetro para a função *hash*, a qual retorna o *hash* correspondente ao parâmetro de entrada. O *hash* é então usado como um identificador único para o arquivo e serve como um apontador para o nodo responsável pelo seu armazenamento.

A função *hash* de uma DHT deve ser elaborada de maneira a atribuir um identificador único para diferentes objetos. Quando dois ou mais objetos distintos recebem o mesmo identificador ocorre uma colisão, impossibilitando a diferenciação entre estes objetos. A solução comumente utilizada para evitar as colisões é adotar um espaço de identificadores grande o suficiente, a ponto que a probabilidade de dois objetos receberem a mesma chave torne-se praticamente nula.

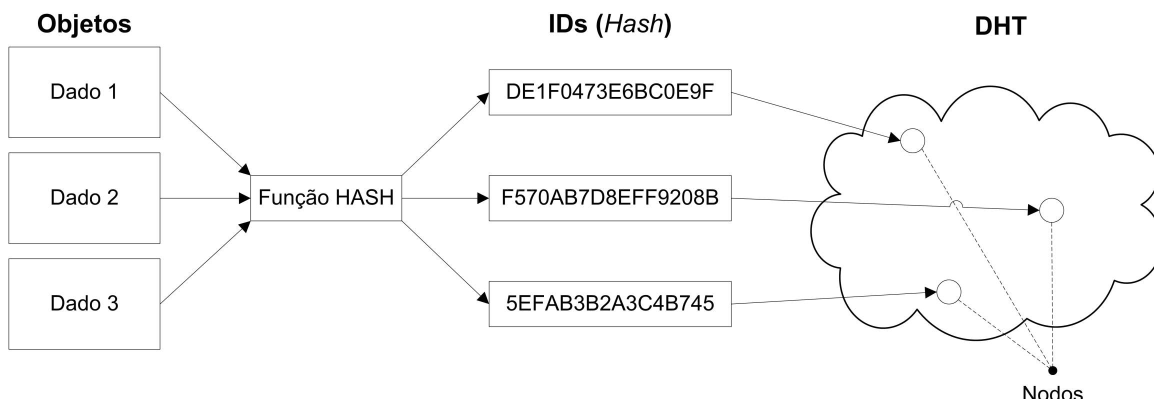

Figura 2.2: Geração das chaves (IDs) e armazenagem de objetos em uma DHT.

Tipicamente a função *hash* empregada pelas DHTs retorna uma cadeia de caracteres de 160 bits baseada em um algoritmo criptográfico (p.ex., SHA1 [28]). A figura 2.3 mostra os resultados da função SHA-1 para dois conjuntos de dados. O valor *hash* está em representação hexadecimal. Observe que mesmo uma pequena mudança na mensagem, como substituir uma letra 't' por 'd', resulta em um *hash* completamente diferente, o que significa que a função SHA-1 tem um bom efeito avalanche [28].

```
SHA1("A mente que se abre a uma nova idéia jamais voltará ao seu
tamanho original.") = 39f96ccb 05cc7f43 0ed43fb7 f8c99f42 9d8f550c

SHA1("A mente que se abre a uma nova idéia jamais voltará ao seu
damanho original.") = 4cec213b fa77c446 4bd9ee5c 35e24307 ae32f34b
```

Figura 2.3: Resultado da função criptográfica e efeito avalanche da função criptográfica SHA-1.

O espaço de identificadores, mais conhecido como espaço de chaves, é definido pela faixa dos possíveis valores numéricos gerados pela função *hash*. As DHTs utilizam um esquema de particionamento para dividir a posse do espaço de chaves entre os nodos participantes do sistema. O particionamento, em geral, emprega alguma variante de *hash* consistente [19], de forma a mapear um sub-conjunto do espaço de chaves para cada um dos nodos.

Os *hashes* consistentes têm a propriedade de exigir atualizações somente nos nodos adjacentes ao nodo que originou um evento de entrada, saída ou falha. Desta maneira,

nenhum dos demais nodos do sistema são afetados, diferentemente do que ocorre em uma tabela *hash* tradicional na qual a adição ou remoção de um dado implica em remapear o espaço de chaves praticamente por completo. Considerando o fato de que qualquer mudança na posse de chaves, usualmente corresponde à transferência dos objetos alocados em um nodo para outro, minimizar essa reorganização é fundamental para suportar *churn*. Além disso, o uso de *hash* consistente em conjunto com o efeito avalanche tem a tendência de balancear a carga do sistema, no sentido que cada nodo irá armazenar praticamente a mesma quantidade de objetos.

O esquema de particionamento emprega uma função $\delta(k_1, k_2)$, a qual define uma noção abstrata de distância entre as chaves k_1 e k_2, de forma que um nodo cuja chave é i fica responsável por todas as chaves mais próximas de i, medido de acordo com a função δ. A implementação da função δ pode ser realizada de diversas formas, como por exemplo: o Chord [15] que usa a diferença matemática entre as chaves; o Pastry [33] e o Tapestry [37] que usam o número de bits em comum entre as chaves; e o Kademlia [26] que usa a operação OU EXCLUSIVO *(XOR)* da lógica binária.

A figura 2.4 exemplifica o particionamento do espaço de chaves para uma DHT Chord. No Chord o espaço de chaves é particionado em segmentos contínuos, nos quais os valores das extremidades correspondem aos identificadores dos nodos pertencentes ao sistema. Se i_1 e i_2 são dois identificadores de nodos adjacentes, então o nodo com ID i_2 é responsável por todas as chaves entre i_1 e i_2.

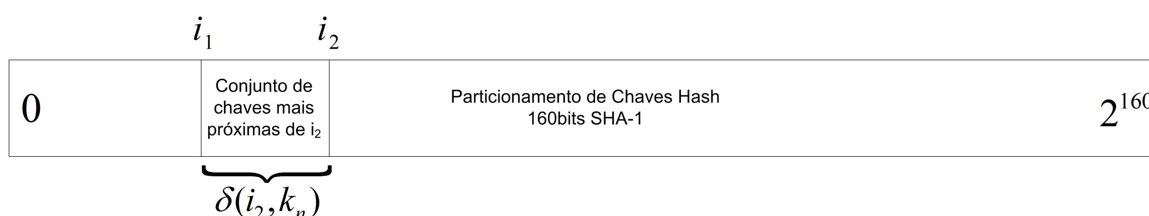

Figura 2.4: Exemplo do particionamento do espaço de chaves utilizado pelo Chord.

Os nodos participantes da DHT devem manter uma tabela de referência para outros nodos, na qual cada entrada associa um nodo a sua chave e ao seu endereço de rede. A operação de consulta $lookup(K)$ faz uso desta tabela, também chamada de tabela de

roteamento, para encontrar o nodo responsável por uma determinada chave. A função $lookup(K)$ retorna a referência para o nodo responsável pela chave K, conforme as regras de particionamento empregadas pela DHT, viabilizando a implementação das duas funções que mais caracterizam as DHTs: a inserção, representada por $put(key, value)$ e a recuperação, representada por $value = get(key)$.

O tamanho da tabela de roteamento depende da implementação da DHT e tem impacto direto na latência de uma consulta e na demanda de recursos de rede para manutenção das tabelas. A latência é expressada pelo número de *hops* (saltos) necessários para que uma consulta seja resolvida, ou seja, cada consulta percorrerá diversos nodos até alcançar um que seja capaz de fornecer a resposta.

O tamanho da tabela de roteamento impacta diretamente na relação entre latência da consulta e o consumo de banda de rede, uma vez que, quanto maior for a tabela de roteamento, menor deverá ser a latência, mas maior será a demanda de banda de rede para sua manutenção [35]. A quantidade mínima de entradas para a tabela de roteamento é de uma referência e a quantidade máxima é de N referências, onde N é o número total de nodos da DHT.

As DHTs que mantêm tabelas de roteamento com N entradas são conhecidas por *Single Hop DHTs* pois a sua função *lookup* consegue resolver qualquer consulta usando apenas a tabela local. Esta abordagem requer que os eventos de entrada, saída ou falha de qualquer nodo do sistema sejam reportados para todos os participantes sem falha [1].

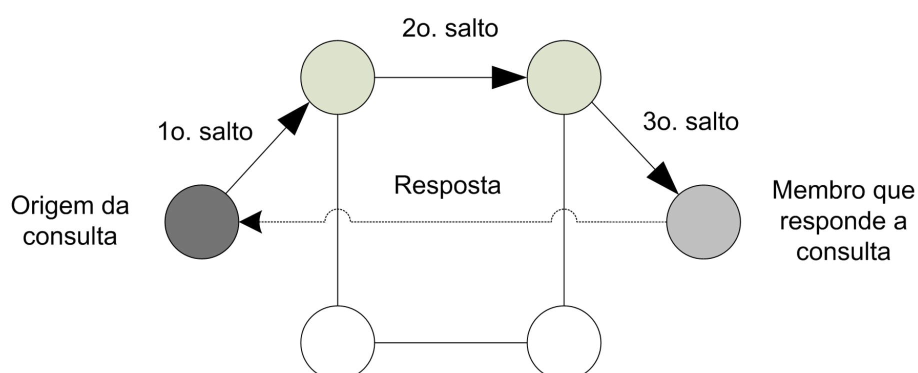

Figura 2.5: Exemplo de roteamento de uma consulta em uma DHT estruturada.

A grande maioria das DHTs, entre elas o Chord, CAN e Tapestry, optaram por implementar a tabela de roteamento com um número total de entradas reduzida. Nestes casos, o algoritmo de roteamento distribuído para as consultas deve explorar o conhecimento dos participantes da rede para localizar o objeto de interesse e garantir que qualquer participante consiga acessar qualquer objeto na DHT, conforme ilustra a figura 2.5. Na ilustração, o nodo que deseja fazer uma consulta, primeiramente verifica sua própria tabela de roteamento, não encontrada a resposta ele repassa a consulta para um de seus vizinhos e este, por sua vez, realiza o mesmo procedimento. A consulta segue de nodo em nodo até encontrar um que seja capaz de fornecer a resposta.

Nas próximas duas seções serão apresentados alguns modelos de sistema DHT, primeiramente alguns exemplos de DHTs de múltiplo saltos *(multi-hop)* e em seguida alguns exemplos de DHTs de salto único *(single-hop)*.

2.3 Tabelas Hash Distribuídas de Múltiplos Saltos

Devido ao compromisso entre a latência de uma consulta e o consumo de recursos de rede necessários para manter a tabela de roteamento, a maioria das DHTs propostas optaram por soluções onde as consultas são resolvidas em múltiplos saltos (p.ex., [11, 18, 5, 26, 31, 33, 15, 37]), de maneira a minimizar o tráfego destinado para manutenção do sistema. Deve-se levar em consideração que na época em que estas DHTs foram propostas o principal desafio era o suporte a sistemas de grande escala e altamente dinâmicos, que incluíam computadores domésticos com conexões instáveis e de baixa banda passante, como as conexões *dial-up*.

Nos tópicos seguintes serão descritas três DHTs de múltiplos saltos: CAN, Chord e Tapestry. A escolha destes sistemas se deu principalmente pelas particularidades que os diferem um do outro, bem como pela sua relevância.

2.3.1 CAN

O CAN *(Content Addressable Network)* [31] tem sua arquitetura baseada em um espaço cartesiano multi-dimensional de coordenadas, o qual é particionado dinamicamente entre todos os participantes do sistema, de forma que cada nodo tem sua própria região dentro do espaço total. Os participantes de um sistema CAN mantêm uma tabela de roteamento com os endereços IPs e as coordenadas virtuais de todos os seus vizinhos. O CAN tem performance de roteamento de $O(d.N^{\frac{1}{d}})$ saltos para uma tabela de roteamento de $2.d$ entradas, onde N é o total de participantes e d é a quantidade de dimensões do espaço de coordenadas.

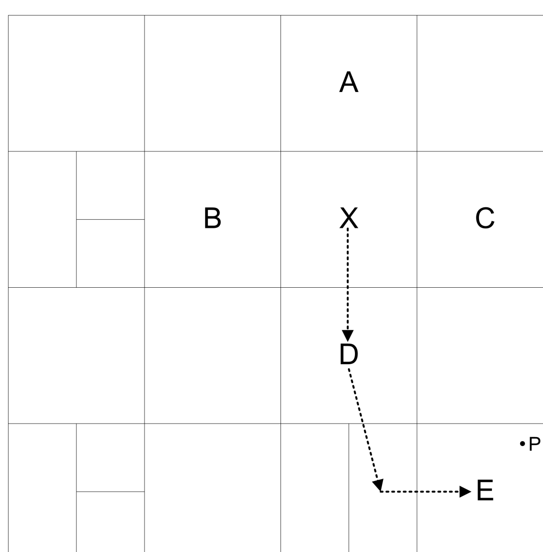

Figura 2.6: Exemplo de roteamento do nodo X para o nodo E em um sistema CAN.

O espaço de coordenadas virtual é usado para armazenar todos os pares $(chave, valor)$ entre os participantes da rede, onde a chave K corresponde a um ponto P no espaço de coordenadas. Para recuperar um valor armazenado no sistema, a operação *lookup* inicialmente deve executar a função *hash* para descobrir a chave K associada ao valor. Quando o nodo que recebe uma mensagem de consulta cujo endereço de destino não pertença a ele, este deve reencaminhar a mensagem para seu vizinho cuja coordenada é a mais próxima daquela especificada na mensagem. Para isso, os participantes do sistema mantêm os endereços IPs dos nodos cuja região de coordenada é contígua a sua própria região.

Conforme mostra a figura 2.6, adaptada do artigo CAN [31], o conhecimento dos vizinhos imediatos no espaço de coordenadas permite o roteamento eficiente entre pontos neste espaço. Considere uma consulta iniciada em X por uma chave qualquer cujo ponto P está localizado na região de E. X primeiramente verifica sua própria tabela de roteamento para determinar qual de seus vizinhos é o mais próximo de P, neste caso é o nodo D. Então, X encaminha a consulta para D e o procedimento de roteamento se repete através da infraestrutura CAN até que a consulta atinja o nodo E.

Considerando que existem diversos caminhos possíveis entre dois pontos no espaço, quando um ou mais vizinhos a um determinado nodo falham, este ainda consegue rotear suas mensagens através do próximo melhor caminho disponível. Para um espaço $d-$*dimensional* particionado em n regiões iguais, a distância média entre um par de nodos qualquer é de $\frac{d}{4}.(n^{\frac{1}{d}})$ saltos [31].

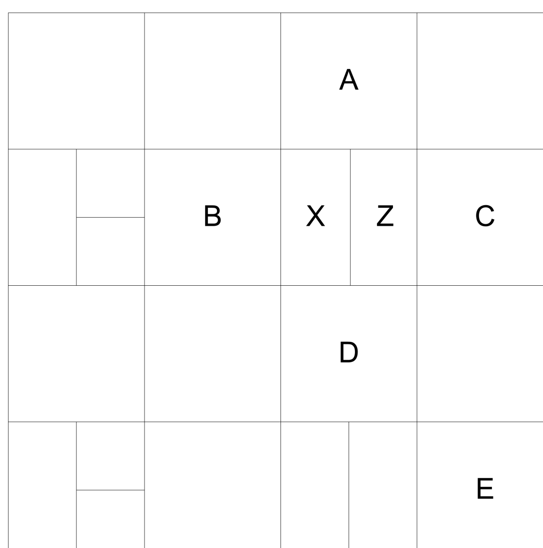

Figura 2.7: Exemplo da entrada do nodo Z em um sistema CAN de 2 dimensões.

Um novo nodo que entra em um sistema CAN precisa ter para si uma parte do espaço de coordenadas. Isso é realizado dividindo-se o espaço de algum outro nodo pela metade, mantendo-se metade para o nodo original e alocando-se a outra metade para o novo nodo. Para entrar no sistema, um novo nodo precisa conhecer de ante-mão o endereço IP de algum nodo já incluso. Satisfeita esta condição, o novo nodo escolhe de forma randômica um ponto P qualquer no espaço de coordenadas e envia uma requisição *join* destinada para

P através do nodo conhecido. A mensagem *join* é então roteada entre os participantes da rede CAN até chegar no nodo da região que contém P. Este nodo, na sequência, divide seu espaço pela metade e associa uma das metades para o novo nodo. A figura 2.7 ilustra a entrada do nodo Z na rede CAN que anteriormente estava sendo representada pelo figura 2.6. Quando um nodo sai da rede CAN, um algoritmo de saída garante que um dos nodos vizinhos àquele que saiu assuma para si o espaço que foi deixado.

2.3.2 Chord

O Chord faz uso de *hash* consistente para associar chaves aos *peers* (nodos) através da função *hash* SHA-1 [28]. O identificador de um *peer* é gerado através do *hash* de seu endereço IP, enquanto que a chave de um objeto é produzida pelo *hash* dos dados do objeto. O conjunto dos possíveis valores produzidos pela função *hash* SHA-1 forma o espaço de identificadores, os quais são distribuídos de forma ordenada e balanceada em um círculo, chamado de *Chord Ring*. Uma chave K é associada ao primeiro *peer* cujo identificador é igual ou seguinte a K no espaço de identificadores. Este participante é chamado de *peer* sucessor da chave K, denotado por $sucessor(K)$, de forma que o sucessor de uma chave K qualquer é sempre o primeiro *peer* no sentido horário a partir de K no *Chord Ring*.

Na figura 2.8 (adaptado de [15]) o *Chord Ring* está sendo representado com 10 *peers* e armazena 5 chaves. Neste exemplo, o sucessor da chave K10 é o *peer* N14, sendo este o nodo responsável por armazenar o valor correspondente àquela chave.

Um novo *peer* X, para entrar no *Chord Ring*, precisa executar um procedimento de entrada que envolve primeiramente descobrir qual é o seu nodo sucessor S e remapear algumas chaves associadas a S para o novo *peer* X. Para exemplificar, considere que um novo *peer* entra no sistema ilustrado pela figura 2.8 com identificador N26. Este novo *peer* fica responsável pela faixa de chaves entre K22 até K26. Quando um *peer* Z sai do sistema Chord, todas as suas chaves são reassociadas ao sucessor de Z.

Cada *peer* do sistema Chord precisa saber como contactar seu sucessor atual no anel de

identificadores. Uma requisição *lookup* envolve comparar a chave com o identificador do *peer* e reencaminhar a requisição pelo *Chord Ring* através dos *peers* sucessores, até que se encontre um par de *peers* cujo valor da chave está contido nos valores dos identificadores deste par; o segundo *peer* deste par é o *peer* que responde à requisição. A figura 2.8 ilustra este procedimento, onde o *peer* $N8$ realiza um *lookup* pela chave $K54$. Primeiramente $N8$ invoca a operação *findsucessor(K54)*, a qual retorna o sucessor desta chave, neste exemplo o *peer* $N55$. A consulta é então roteada pelos *peers* no anel entre o *peer* $N8$ e o *peer* $N55$. A resposta é retornada através do caminho inverso.

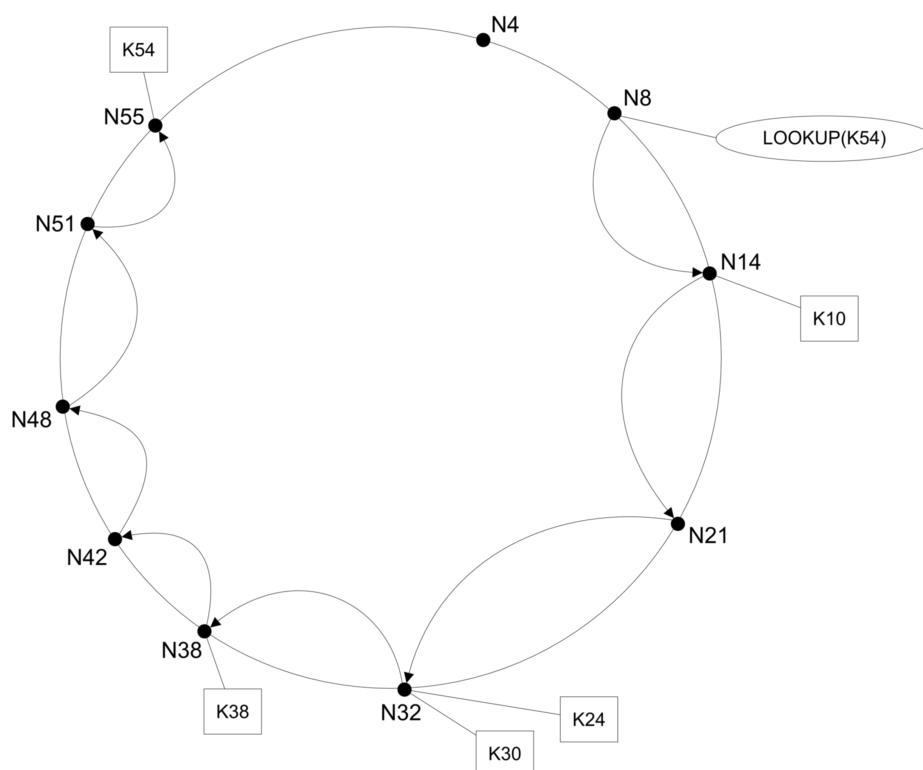

Figura 2.8: Representação de um sistema Chord com 10 participantes e 5 objetos.

Para diminuir a latência das consultas, o Chord mantém uma tabela de roteamento chamada de *finger table*, na qual a quantidade de entradas m é igual à quantidade de bits da chave gerada pela função *hash*. A *i-ésima* entrada da *finger table* do *peer* N contém o identificador do primeiro *peer* S que sucede N por pelo menos 2^{i-1} no anel de identificadores, por exemplo $S = successor(N + 2^{i-1})$, onde $1 \leq i \leq m$, logo S é a *i-ésima* entrada da *finger table* de N ($S = N -> finger[i]$). Uma entrada na *finger table* é composta pelo identificador Chord e pelo endereço IP (com o número de porta) para o

respectivo *peer*. A figura 2.9 mostra a *finger table* do *peer* $N8$ e os respectivos *peers* para os quais cada uma das entradas aponta.

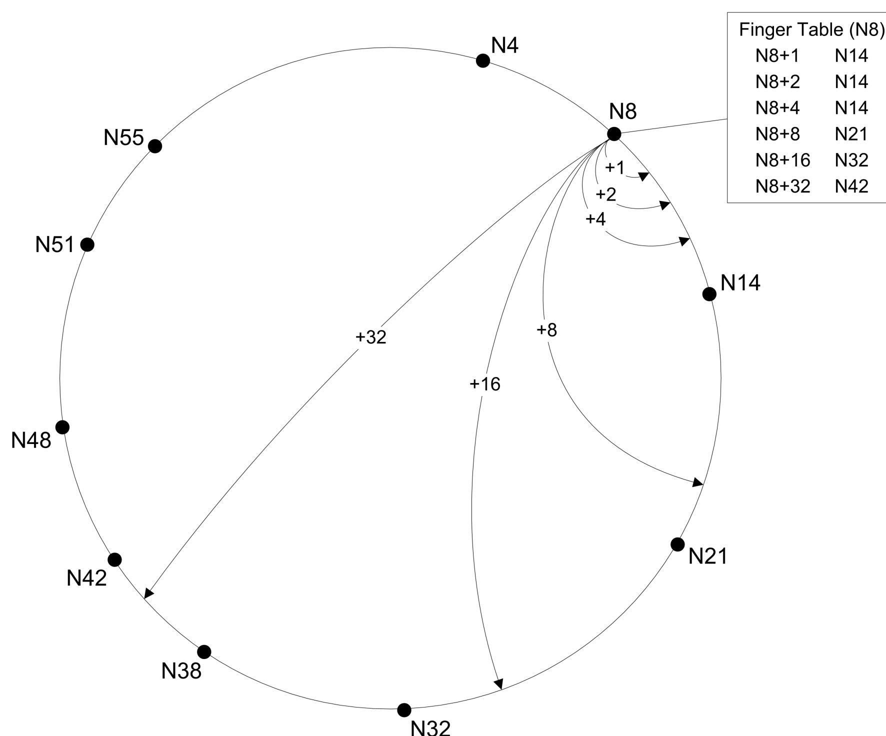

Figura 2.9: Exemplo de uma Finger Table para *peer* 8.

Quando um novo *peer* X entra no sistema, o ponteiro sucessor do *peer* antecessor a X precisa ser atualizado. Para isso, o Chord faz uso de um algoritmo de estabilização que roda periodicamente em segundo plano e tem o objetivo de atualizar os ponteiros sucessores para as entradas das *finger tables*. A exatidão do protocolo Chord baseia-se no fato que cada *peer* tem conhecimento preciso de seu sucessor. Quando um participante falha, pode acontecer que algum *peer* fique sem ter conhecimento sobre seu novo sucessor, pois ainda não teve chance de aprender sobre isso. Para contornar essa situação, os *peers* mantêm uma lista de sucessores de tamanho r, a qual contém os primeiros r sucessores do *peer*. Quando o sucessor não responde, o *peer* simplesmente contacta o próximo *peer* de sua lista de sucessores.

2.3.3 Tapestry

A arquitetura Tapestry [37] é baseada na técnica de busca distribuída de Plaxton e na estrutura de dados distribuídos *Plaxton Mesh* [30], com mecanismos adicionais para prover disponibilidade, escalabilidade e auto-configuração, a fim de suportar *churn*.

Em uma DHT Tapestry cada nodo mantém uma tabela de roteamento para seus vizinhos, chamada de "mapa de roteamento local", conforme mostra a figura 2.1, adaptada de [2]. Este mapa possui múltiplos níveis, sendo que cada nível é representado por uma determinada quantidade de dígitos, ou seja, o nível l conterá ponteiros para os nodos cujo ID coincidir com os últimos l dígitos. A distribuição e busca dos objetos em um sistema Tapestry é baseada na aproximação iterativa dos identificadores.

Cada posição da tabela corresponde a um
ponteiro para um nodo na rede Tapestry.

	Nível 5	Nível 4	Nível 3	Nível 2	Nível 1
Entrada 0	07493	x0493	xx093	xxx03	xxxx0
Entrada 1	17493	x1493	xx193	xxx13	xxxx1
Entrada 2	27493	x2493	xx293	xxx23	xxxx2
Entrada 3	37493	x3493	xx393	xxx33	xxxx3
Entrada 4	47493	x4493	xx493	xxx43	xxxx4
Entrada 5	57493	x5493	xx593	xxx53	xxxx5
Entrada 6	67493	x6493	xx693	xxx63	xxxx6
Entrada 7	77493	x7493	xx793	xxx73	xxxx7
Entrada 8	87493	x8493	xx893	xxx83	xxxx8
Entrada 9	97493	x9493	xx993	xxx93	xxxx9

Tabela 2.1: Mapa de vizinhos mantido pelo nodo 67493 [37].

$$xxxx7 \rightarrow xxx67 \rightarrow xx567 \rightarrow x4567 \rightarrow 34567$$

Figura 2.10: Sequência de resolução dígito por dígito do Tapestry.

Para exemplificar, considere o terceiro nível da entrada 5 para o mapa de roteamento do nodo 67493 da figura 2.1. Esta posição contém um ponteiro para o nodo mais próximo de 67493 na rede, cujo ID termina em xx593. As mensagens são roteadas entre os participantes de forma incremental, dígito por dígito, até que chegue ao nodo de destino. A figura 2.11 mostra um exemplo do caminho realizado por uma mensagem partindo do

nodo com ID igual a 67493 para o nodo de ID igual a 34567. Os dígitos são resolvidos da direita para a esquerda, conforme a figura 2.10.

Quando a tabela de roteamento de um nodo não possuir a entrada para um participante que compartilhe o sufixo comum com a chave (em pelo menos um ou mais dígitos), a mensagem é encaminhada para o nodo que está numericamente mais próximo da chave em relação ao nodo atual.

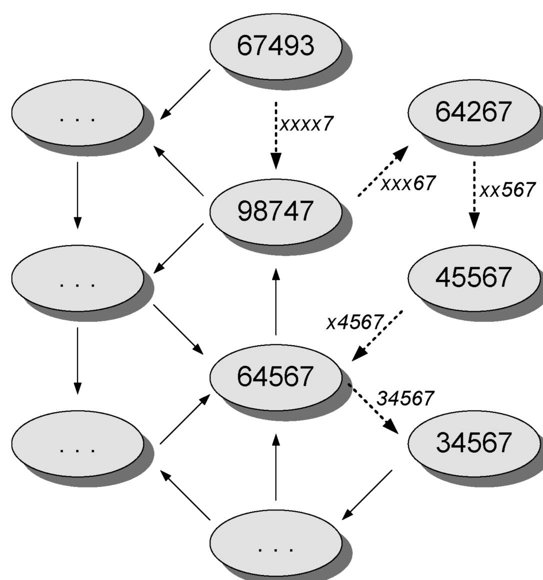

Figura 2.11: Exemplo do caminho de roteamento de uma mensagem em uma DHT Tapestry [37].

Este método de roteamento garante que qualquer nodo existente no sistema possa ser localizado dentro de no máximo $log_B N$ saltos, onde N é o número total de nodos no sistema e B é a base numérica usada para representar os identificadores, nos exemplos usamos base decimal ($B = 10$). Desta forma, um nodo precisa manter apenas uma tabela com B entradas para cada nível de roteamento.

2.4 Tabelas Hash Distribuídas de Salto Único

As DHTs de salto único têm como principal compromisso minimizar ao máximo a latência das consultas, mas sem negligenciar as questões relativas ao consumo de recursos de rede para a manutenção do sistema. O foco principal passa a ser como realizar a disseminação

dos eventos de entrada e saída de participantes de maneira rápida e eficiente [16], visto que a realização de consultas em DHTs de salto único é trivial pois cada um dos nodos têm conhecimento sobre todos participantes do sistema.

As próximas subseções apresentam dois trabalhos relacionados às DHTs de salto único, primeiramente a OneHop DHT e em seguida a D1HT.

2.4.1 OneHop DHT

O sistema OneHop [1] foi a primeira DHT proposta a garantir que a maior parte das consultas sejam resolvidas em um único salto, mesmo em ambientes dinâmicos. A topologia da rede de sobreposição adotada pelo OneHop é similar àquela empregada pelo Chord, onde os nodos são distribuídos e ordenados por seus identificadores em um anel, mas com a diferença que cada nodo, além de conhecer seu sucessor, também conhece seu antecessor. A função *hash* utilizada para o mapeamento das chaves e para geração de identificadores dos participantes é o SHA-1 de 128bits.

Neste sistema, a propagação dos eventos de entrada e saída de nodos é realizada através de uma estrutura hierárquica, cujo objetivo é viabilizar a consolidação de várias notificações de eventos em uma mesma mensagem, a fim de minimizar o uso de recursos de rede para manutenção das tabelas de roteamento. A hierarquia definida pelo OneHop DHT agrupa os participantes em *slices*, os quais possuem uma divisão interna em grupos de tamanho igual chamados *unidades*. Cada *slice* e cada *unidade* tem um líder, de forma que os nodos são caracterizados por três níveis distintos: nodos comuns, líderes de *slices* e líderes de unidades. Estes componentes são descritos a seguir.

Os líderes de *slice* são responsáveis por reunir as informações sobre todos os eventos ocorridos dentro de seu próprio *slice* e, periodicamente, propagar este pacote de informações aos outros líderes de *slice*. A figura 2.12 ilustra este processo: considere o nodo rotulado por X, quando este detecta um novo evento envia uma mensagem de notificação para seu líder de *slice* (Passo 1). Este líder recolhe todas as notificações que recebe de seu próprio *slice* e consolida em uma única mensagem para enviar aos demais líderes de

slice do sistema (Passo 2). Ao receber esta mensagem, os líderes de *slice* a propagam para seus líderes de *unidade* (Passo 3) que, por sua vez, propagam aos nodos comuns (Passo 4). Uma descrição mais detalhada sobre o fluxo de notificação de eventos em um sistema OneHop pode ser encontrada em [1].

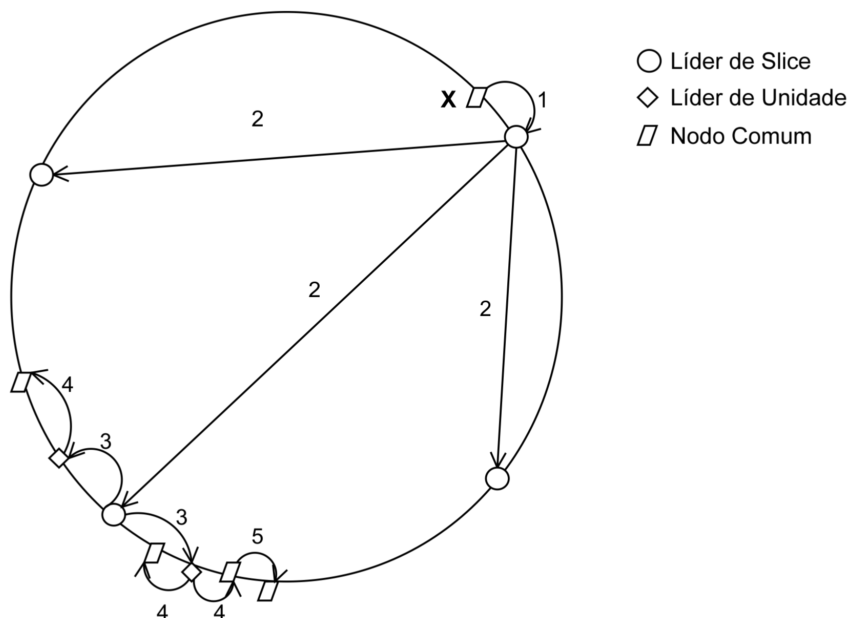

Figura 2.12: Fluxo das notificações de eventos em um sistema OneHop [1].

A hierarquia definida por OneHop facilita o agrupamento de eventos de maneira a minimizar os custos de manutenção das tabelas de roteamento, mas tem como efeito colateral um alto nível de desbalanceamento de carga de manutenção entre os nodos de diferentes níveis da hierarquia, como mostrado nos resultados publicados em [29]. Além disso, os próprios autores apontam limitações de escalabilidade nos casos onde um líder de *slice* possui uma banda de rede menor daquela exigida para este cargo.

2.4.2 D1HT

A D1HT [23] propõe resolver grande parte das consultas em apenas um único salto, com baixa demanda de rede e em ambientes dinâmicos típicos das aplicações P2P. Os participantes e objetos de um sistema D1HT recebem um identificador (ou uma chave no caso dos objetos) através da função criptográfica SHA-1, e são dispostos ordenadamente em um anel, de forma similar ao que acontece em um sistema Chord. As chaves, como

também ocorre no Chord, são associadas ao nodo sucessor no sentido horário, na topologia formada pelo anel.

Os nodos mantêm uma tabela de roteamento com os endereços IP de todos os participantes do sistema para viabilizar que as consultas sejam resolvidas em apenas um salto. Para manter a tabela de roteamento atualizada e sincronizada em todos os nodos, a D1HT utiliza um algoritmo de disseminação de eventos batizado de EDRA (*Event Detection and Report Algorithm*). O EDRA faz uso de uma tabela adicional que possui $log_2 N$ referências para outros nodos. Cada entrada desta tabela aponta para um nodo cuja distância, em sentido horário em relação ao nodo, é função de 2^i, onde i é o índice da entrada e começa em 0. Esta operação resulta em uma tabela cujas distâncias para os nodos crescem em razão logarítmica. A construção desta tabela se dá de forma parecida à *Finger Table* empregada pelo Chord, com a diferença que no EDRA a tabela tem a finalidade de auxiliar na disseminação de eventos e não de diminuir a latência das consultas *lookup*, como ocorre no Chord.

Os nodos são responsáveis por monitorar seu predecessor na rede virtual e reportar os eventos detectados para seus sucessores. O EDRA viabiliza que todos os nodos do sistema terão conhecimento de um novo evento em tempo logarítmico. A disseminação é realizada com o auxílio da tabela de referências descrita acima e com o emprego de mensagens com tempo-de-vida (TTL). A figura 2.13 demonstra este procedimento para um sistema com 11 nodos, onde o nodo P_P falha e este evento é detectado por seu sucessor P. P irá disseminar este evento através de 4 mensagens endereçadas aos nodos em sua tabela de referências P_1, P_2, P_4, P_8, representadas pelas setas contínuas. Os nodos P_2, P_4 e P_8 irão decrementar o TTL da mensagem e reencaminhar para os nodos de suas respectivas tabelas, cujo log base dois da distância seja igual ou menor que o TTL da mensagem, conforme indicado pelas setas tracejadas da figura. Como P_6 irá receber a mensagem com $TTL = 1$, irá repetir o procedimento e reencaminhar para P_7 (seta pontilhada).

Em [23] o autor enfatiza o baixo consumo de recursos de rede demandados para manutenção das tabelas de roteamento se comparado com outros sistemas tipo *single hop*

DHT, como por exemplo [1]. O autor descreve, ainda, algumas situações em que uma consulta pode levar mais tempo do que o esperado em decorrência de um evento ainda não ter sido totalmente propagado no sistema.

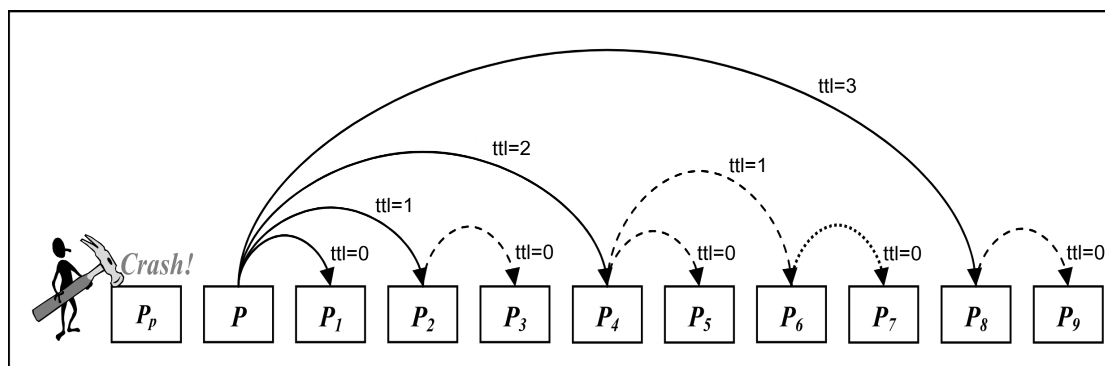

Figura 2.13: Disseminação de eventos em um sistema D1HT [23].

CAPÍTULO 3

ALGORITMOS DE DIAGNÓSTICO DISTRIBUÍDO

Este capítulo traz, primeiramente, as definições preliminares sobre algoritmos de diagnóstico distribuído, seguido pela classe de algoritmos adaptativos de diagnóstico distribuído. O capítulo prossegue com uma discussão sobre algoritmos hierárquicos de diagnóstico distribuído e é finalizado com a apresentação do algoritmo DiVHA e da rede HyperBone.

3.1 Definições Preliminares

Os algoritmos de diagnóstico distribuído têm como objetivo determinar de maneira precisa o estado (falho ou não-falho) de todos os participantes de um sistema distribuído [9, 13, 36]. Esta classe de algoritmos pode ser aplicada em sistemas de gerenciamento de falhas, pois o diagnóstico em nível de sistema é uma abordagem eficaz para construção de sistemas distribuídos e de monitoração de rede tolerantes a falhas [27, 14, 3].

Considere um sistema com N nodos onde cada nodo pode estar em um de dois estados, falho ou não-falho. E, que existe um enlace de comunicação entre qualquer par de nodo do sistema e esses enlaces nunca falham. Os nodos em um sistema de diagnóstico são capazes de testar outros nodos e determinar seu estado corretamente. O conjunto de testes assinalados para cada um dos nodos, bem como a realização do diagnóstico a partir do resultado dos testes, são funções elementares de um algoritmo de diagnóstico distribuído [10].

Os algoritmos de diagnóstico, usualmente, modelam o sistema por meio de um grafo direcionado $G = (V, E)$, onde os vértices do grafo representam os nodos do sistema e as arestas indicam os testes que são realizados. A figura 3.1 mostra um exemplo de sistema

representado por um grafo com 4 nodos, o vértice na cor preta indica que o nodo está falho. Uma aresta saindo do nodo u para o nodo v significa que o nodo u testa o nodo v. Cada teste corresponde a um procedimento cujo resultado classificará o nodo testado em falho ou não-falho.

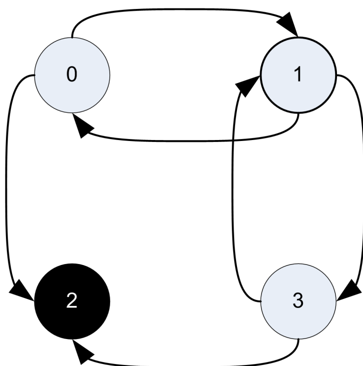

Figura 3.1: Um exemplo de assinalamento de testes; o nodo 2 está falho.

O primeiro modelo de diagnóstico foi o PMC, cujo nome deriva das iniciais dos autores [10]. O modelo PMC assume a existência de um observador central que coleta os resultados dos testes realizados pelos participantes do sistema e analisa esses resultados para determinar o estado de cada um dos nodos. Os testes entre os nodos são fixados previamente e o conjunto de todos os testes é o que possibilita que o diagnóstico do sistema seja realizado de forma correta. O modelo ainda assume que os participantes executam os testes com 100% de precisão e reportam os resultados para o observador central de forma correta. A coleção dos resultados dos testes é chamada de *síndrome* do sistema. Além do diagnóstico, os autores também definem a *diagnosticabilidade* da seguinte maneira: um sistema é *t-diagnosticável* se dada uma síndrome, o supervisor central completa o diagnóstico caso o número de unidades falhas não exceda t.

No começo dos anos 80, uma nova abordagem de diagnóstico foi proposta por Kuhl e Reddy [20], onde participantes do sistema realizam o diagnóstico de maneira independente. O algoritmo *New-Self* [20], proposto pelos autores, foi o primeiro a realizar o diagnóstico de maneira distribuída, no qual um conjunto fixo de testes é previamente atribuído para cada nodo e os resultados dos testes são compartilhados entre os nodos não-falho. Apesar do mérito do desenvolvimento de uma solução distribuída de diagnóstico que elimina a

necessidade de um observador central, o algoritmo *New-Self* é considerado custoso [10].

3.2 Diagnóstico Adaptativo e Distribuído

O próximo passo na evolução dos algoritmos de diagnóstico aconteceu no começo dos anos 90 com o Adaptive-DSD [3], cuja abordagem permitiu que o diagnóstico fosse realizado de maneira distribuída e adaptativa, ou seja, sem necessitar de uma unidade central e com a configuração dos testes determinada dinamicamente, baseada nos resultados dos testes executados anteriormente. No algoritmo Adaptive-DSD cada nodo recebe um identificador numérico e são dispostos sequencialmente formando um anel. Os nodos devem testar seus sucessores até encontrar um que não esteja falho, do qual colhe dados para atualizar as informações de diagnóstico.

O Adaptive-DSD, bem como os demais algoritmos de diagnóstico distribuído, podem ser avaliados por duas métricas: (1) a quantidade de testes necessários por rodada; (2) a latência de propagação de um evento. Um evento é definido pela mudança de estado de um nodo, de não-falho para falho ou vice-versa. A latência é expressa em rodadas de testes e corresponde ao tempo necessário para que todos os nodos não-falho descubram a ocorrência de um novo evento [22]. Uma rodada de testes é definida como o período de tempo necessário para que todos os nodos em estado não-falho executem seus testes. Quanto menor forem esses valores (quantidade de testes e latência) mais eficiente é o algoritmo.

A figura 3.2 mostra um sistema executando o algoritmo Adaptive-DSD. Os nodos em preto estão falhos, enquanto os demais estão em estado não-falho. Em uma rodada de testes cada nodo testa sequencialmente seus sucessores como, por exemplo, o nodo 0 que testa o nodo 1 e descobre que este está falho; o nodo 0 então testa o nodo 2, não-falho e recebe informações de diagnóstico deste. De forma análoga comportam-se os nodos 2, 3, 6 e 7.

No que diz respeito à quantidade de testes, o algoritmo Adaptive-DSD é considerado muito eficiente pois executada apenas N testes por rodada, sendo o mínimo necessário

para que o diagnóstico seja corretamente determinado. Quanto à latência, podem ser necessárias até N rodadas de testes para que todos os nodos tomem ciência de um determinado evento. Isto ocorre porque os eventos são encaminhados para apenas um novo nodo a cada rodada, o que torna o Adaptive-DSD ineficiente neste quesito.

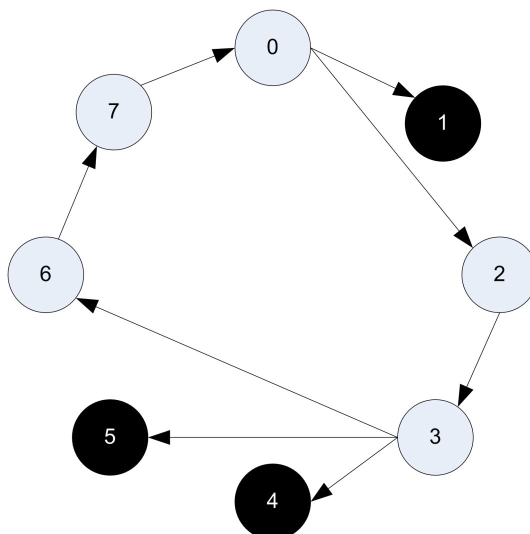

Figura 3.2: Exemplo de uma rodada de testes realizada pelo algoritmo Adaptive-DSD.

A alta latência do Adaptive-DSD pode impossibilitar o diagnóstico nos casos de sistemas altamente dinâmicos (*churn*) ou com um grande número de participantes. Por exemplo, um sistema com 100 participantes onde os nodos realizam seus testes a cada 30 segundos; neste caso um evento de falha pode demorar até 50 minutos para ser conhecido por todos os nodos. Durante este tempo o nodo que falhou já pode ter sido corrigido e outros eventos podem ter ocorrido. O diagnóstico realizado pelos nodos dificilmente seria uma representação fiel à condição do sistema.

Para diminuir a latência da propagação de eventos podemos optar por uma estratégia hierárquica para o assinalamento de testes e disseminação dos eventos, conforme apresentado a seguir.

3.3 Algoritmos Hierárquicos de Diagnóstico Distribuído

A primeira proposta de diagnóstico distribuído a adotar uma abordagem hierárquica foi o algoritmo Hi-ADSD [17], o qual faz uso do conceito de *clusters* para estabelecer uma

hierarquia de testes entre os nodos. No Hi-ADSD os nodos são agrupados em diversos *clusters*, cujos tamanhos são definidos pelo índice s e determinado por uma potência de 2 (2^s). Um mesmo nodo pode pertencer a *clusters* de índices s diferentes. Um *cluster* de índice $s = 3$ tem tamanho igual a 8 (2^3) e é composto por dois *clusters* de índice 2; e por conta da recursividade, contém quatro *clusters* de índice 1. A figura 3.3 ilustra como esse agrupamento é realizado para um sistema de 8 nodos.

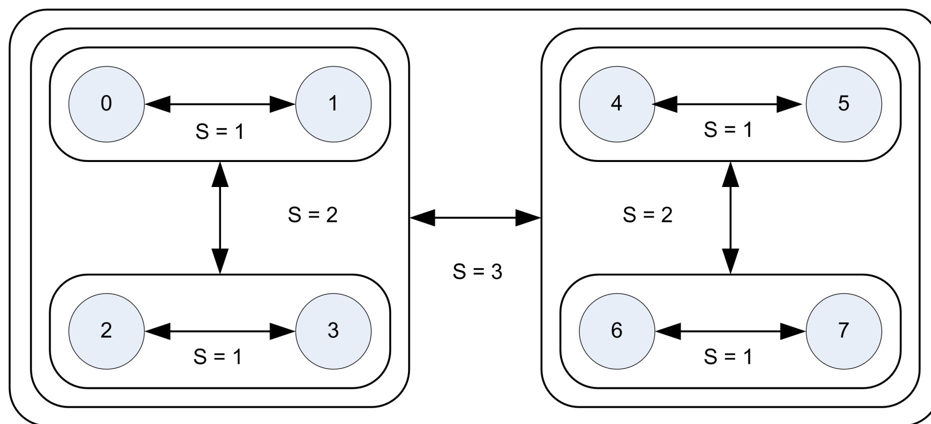

Figura 3.3: Sistema de 8 nodos agrupados em clusters.

Um nodo inicialmente realiza testes em *clusters* de tamanho 2 (2^1) e, nas próximas rodadas, nos *clusters* de tamanho 4 (2^2), 8 (2^3) e assim sucessivamente até atingir o cluster de tamanho $N/2$ ($2^{(log_2 N)-1}$).

A lista dos nodos de um determinado *cluster* é calculada pela função $C(i, s)$ [17] e pode ser definida recursivamente pela expressão: $C_{i,s} = C_{j,s-1} \cup C_{i,s-1}$, $j = i \oplus 2^{s-1}$ e $C_{i,1} = j$, $j = i \oplus 1$. A tabela 3.1 lista os possíveis resultados da $C(i, s)$ para um sistema composto por 8 nodos.

s	$C(0,s)$	$C(1,s)$	$C(2,s)$	$C(3,s)$	$C(4,s)$	$C(5,s)$	$C(6,s)$	$C(7,s)$
1	1	0	3	2	5	4	7	6
2	2,3	3,2	0,1	1,0	6,7	7,6	4,5	5,4
3	4,5,6,7	5,6,7,4	6,7,4,5	7,4,5,6	0,1,2,3	1,2,3,0	2,3,0,1	3,0,1,2

Tabela 3.1: Resultado da função $C(i, s)$ para um sistema de 8 nodos.

Quando um nodo não-falho é testado, fornece informações sobre todos os nodos da $C(i, s)$ ao qual ele pertence. Se nenhum nodo não-falho é encontrado em uma determinada

$C(i, s)$, o nodo i passa a testar $C(i, s+1)$ até que encontre um nodo livre de falhas, ou até descobrir que todos os nodos estão falhos [22]. O número de testes executados pelo Hi-ADSD é de $N^2/4$ e a latência pode chegar, no pior caso, a no máximo $log_2 N$ rodadas de testes.

A busca pela otimização do Hi-ADSD levou este algoritmo a ter duas variantes: o *Hi-ADSD with Detours* e o *Hi-ADSD with Timestamps*. A motivação é encontrar uma solução que diminua a quantidade de testes necessários no pior caso, mas sem aumentar a latência máxima original do Hi-ADSD.

O *Hi-ADSD with Detours* [7] tem como componente principal os desvios (*detours*), que nada mais são que caminhos alternativos no grafo de testes, pelos quais as informações sobre o estado dos nodos podem ser obtidas. A redução da quantidade de testes é obtida através de uma mudança na estratégia de testes: o nodo testador não continua a testar os demais nodos de um *cluster* ao encontrar o primeiro com falha. Quando isso ocorre, o *cluster* é dito bloqueado e o nodo testador procura obter informações sobre os nodos deste *cluster* ao testar outros *clusters*. Testes extras são realizados nos nodos do *cluster* bloqueado apenas quando não for possível receber informações sobre eles por meio dos outros *clusters*. Simulações mostram que o *Hi-ADSD with Detours* [7] necessita de uma quantidade menor de testes em relação ao Hi-ADSD original.

O *Hi-ADSD with Timestamps* [9] é, na realidade, um complemento do *detours*, no qual é aplicado o conceito de que é possível obter informações de diagnóstico sobre um determinado nodo a partir de diversos outros nodos. No entanto, para possibilitar a diferenciação entre uma informação atual de outra desatualizada são utilizados *timestamps*. Um *timestamp* é um contador inicializado em 0 que recebe incrementos na ocorrência de eventos. Números pares indicam que o nodo está não-falho e números ímpares indicam que o nodo está falho. A latência média do algoritmo *Hi-ADSD with Timestamps* é menor em relação ao algoritmo *Hi-ADSD with Detours*. O número máximo de testes é o mesmo do *Hi-ADSD with Detours* visto que a estratégia de testes é a mesma.

3.3.1 O Algoritmo DiVHA e a Rede HyperBone

Esta seção finaliza o presente capítulo apresentando o algoritmo de diagnóstico distribuído hierárquico DiVHA (*Distributed Virtual Hypercube Algorithm*) e a rede HyperBone [22]. O HyperBone é uma rede que oferece serviços de monitoração e roteamento, permitindo a execução de aplicações distribuídas na Internet. A rede de sobreposição empregada pelo HyperBone tem características que permitem implementar um sistema de diagnóstico distribuído.

Neste sistema o algoritmo DiVHA é responsável por gerar o grafo de testes $T(S)$, o qual é utilizado pelo HyperBone para o assinalamento e realização dos testes, bem como para determinar o fluxo de informação no sistema. Na existência de nodos falhos, testes adicionais são incluídos de forma a preservar o diâmetro logarítmico do hipercubo. O algoritmo DiVHA, da mesma forma que o algoritmo Hi-ADSD, utiliza o conceito de *clusters* para agrupar os nodos e organizar os testes. A figura 3.3 ilustra como o agrupamento em *clusters* é estruturado para um sistema com 8 *peers* não-falhos.

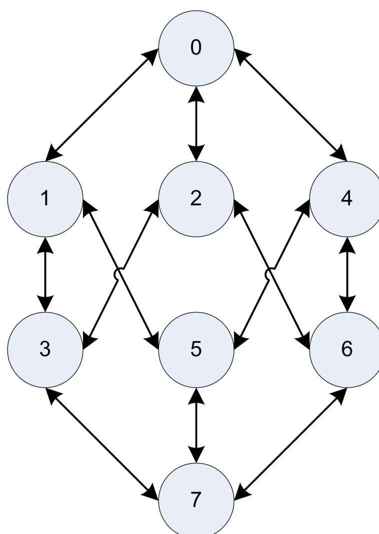

Figura 3.4: Grafo $H(S)$ para um sistema de 8 *peers* não-falhos.

No DiVHA a topologia do sistema é baseada no grafo $H(S)$, conforme ilustra a figura 3.4, onde os vértices de $H(S)$ correspondem aos nodos e as arestas direcionadas representam as conexões virtuais entre eles, este conjunto forma a rede de sobreposição. A distância mínima entre um par de vértices (i, j) qualquer neste grafo é definida por

$h(i, j)$, e é igual ao tamanho do menor caminho entre os vértices i e j em $H(S)$.

O DiVHA pode incluir arestas extras ao grafo $H(S)$ a fim de garantir as propriedades matemáticas do hipercubo virtual que modelam a rede sobreposta, mesmo na ocorrência de falhas de nodos ou quando a distribuição ou a quantidade de nodos do sistema não permita a formação de um hipercubo completo. O resultado da operação do DiVHA é o grafo $T(S)$.

Uma aresta direcionada em $T(S)$ entre os nodos i e j, por exemplo, indica que o nodo i deve manter um enlace virtual com nodo j. Na ocorrência de um evento (entrada, saída, falha ou recuperação de um nodo), todos os nodos do sistema devem recalcular $T(S)$ de forma a determinar, novamente, suas conexões considerando a nova configuração do sistema. Na condição do sistema estar estabilizado $T(S)$ é calculado de forma exatamente igual em todos os nodos.

A construção do grafo $T(S)$ é realizada de maneira a garantir que a distância entre quaisquer nodos em $T(S)$ seja de no máximo $log_2 N$. A manutenção da distância garante a latência máxima. O algoritmo DiVHA verifica progressivamente o atendimento das distâncias mínimas entre os nodos. Inicialmente são verificados os nodos conectados por distância mínima 1, aumentando esta distância até alcançar $log_2 N$. Quando a verificação da distância mínima entre dois nodos falha é inserida uma aresta ligando diretamente estes nodos.

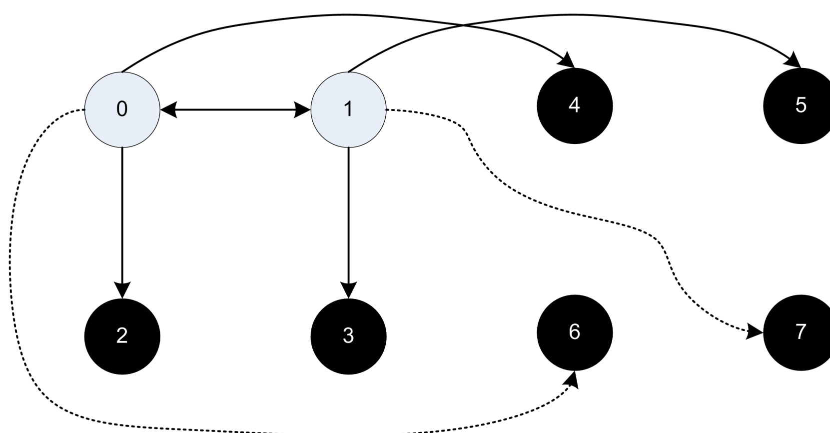

Figura 3.5: Grafo $T(S)$ gerado pelo DiVHA; linha pontilhada indica uma conexão adicional.

A figura 3.5 mostra o grafo $T(S)$ gerado pelo DiVHA para um sistema de 8 nodos, onde os nodos em preto representam nodos em estado falho. A linha pontilhada indica uma conexão adicional, além das definidas por $H(S)$. Estas conexões extras funcionam como atalhos ou pontes, para garantir que a distância máxima entre quaisquer pares de nodos seja de no máximo log_2N.

O pseudo-código do DiVHA é apresentado na figura 3.6. O algoritmo começa sua operação reinicializando $T(S)$ para o estado no qual o número de arestas é 0, conforme indicado pela linha 2. Em seguida, nas linhas 3 a 6, são executados quatro *loops* em cascata, onde cada *loop* varre uma determinada propriedade do modelo do sistema. O laço mais externo percorre progressivamente todos os possíveis índices de *clusters* para o sistema S. O laço seguinte varre todas as possíveis distâncias d em $H(S)$ para o índice de *cluster* corrente.

```
01. DiVHA(system S)
02.    T(S) initially does not have any edge
03.       FOR s = 1 TO log2(N) DO
04.          FOR d = 1 TO s DO
05.             FOREACH fault free node i = [0 ... N-1] DO
06.                FOREACH node j that belongs C(i, s) DO
07.                   IF h(i, j) == d AND (d == 1 OR d(i, j) > s) {
08.                      add edge(i, j) to T(s)
09.                   }
```

Figura 3.6: Algoritmo de assinalamento de testes: DiVHA.

O terceiro *loop* faz uma busca por todos os nodos i não-falho do sistema S e, finalmente, o último *loop* percorre os nodos j pertencentes a $C(i, s)$ atual. Desta forma são verificadas as arestas entre o nodo 0 e nodos pertencentes ao *cluster* $C(0, s)$, entre o nodo 1 e os nodos pertencentes ao *cluster* $C(1, s)$, e assim por diante até o nodo $N-1$ e os nodos pertencentes ao *cluster* $C(N-1, s)$.

A linha 7 verifica as condições necessárias para inclusão de uma nova aresta em $T(S)$ e pode ser lida da seguinte maneira: ***SE*** *a distância de i para j no hipercubo for igual ao d corrente* ***E*** *a aresta corrente pertence a HS(S)* ***OU*** *a distância de i para j no grafo T for maior que o s corrente* ***ENTÃO*** é executada a linha 8. A primeira condicional

assegura que as arestas dos *clusters* de menor índice são verificadas antes que as arestas dos *clusters* de índice maior. A segunda condicional garante que uma aresta é adicionada entre o nodo i e o nodo $j \in C(i, s)$ ou se a aresta pertencer $H(S)$, ou sempre que não existir um caminho do nodo i para o nodo j em $T(S)$ com distância $\leq s$.

A função $d(i, j)$ retorna a distância entre os nodos i e j em $T(S)$ e corresponde ao tamanho do menor caminho entre o nodo i e o nodo j em $T(S)$. Se não existir um caminho entre um par de nodos a distância é tida como infinita.

Satisfeitas as condições, a inclusão da aresta de i para j é realizada pela operação da linha 8 do pseudo-código. Esta nova aresta indica que o nodo i deve estabelecer conexão com o nodo j.

O DiVHA permite que o algoritmo de diagnóstico determine o estado de todos os nodos do sistema em no máximo $log_2 N$ rodadas de testes. O número máximo de testes por rodada é de $N \times log_2 N$. A latência média do algoritmo DiVHA é, em geral, menor que sua latência máxima de $log_2 N$. Resultados experimentais indicam que a maior parte dos eventos são diagnosticados em cerca de $(log_2 N)/2$ rodadas [22].

CAPÍTULO 4

HYPERDHT: DHT DE UM SALTO BASEADA EM HIPERCUBO VIRTUAL DISTRIBUÍDO

Neste capítulo será apresentado e discutido o HyperDHT. As seções estão organizadas da seguinte maneira. A seção 4.1 apresenta algumas considerações preliminares. A seção 4.2 define o modelo do sistema. A seção 4.3 descreve a abordagem adotada para o particionamento do espaço de chaves *hash*. Na seção 4.4 são discutidas as questões relativas às consultas de salto único. A abordagem adotada para replicação de valores é apresentada na seção 4.5. Na seção 4.6 é explicado o funcionamento do protocolo de entrada nas redes HyperDHT, seguido pela seção 4.7, finalizando o capítulo com considerações sobre a saída de participantes de um sistema HyperDHT.

4.1 Considerações Preliminares

Um dos principais requisitos para viabilizar consultas de salto único em DHTs é prover todos os participantes com informações sobre a composição do sistema [1]. Por esta razão, uma DHT de salto único precisa munir-se de meios eficientes para disseminação dos eventos de entrada, saída, falha e recuperação de nodos, a fim de atualizar a visão do sistema em todos os participantes sem falha. As informações sobre os eventos e, portanto, sobre a configuração do sistema, são utilizadas para atualizar a tabela de roteamento dos nodos.

A eficiência do algoritmo para disseminação dos eventos em DHTs depende, entre outros fatores, da estrutura da rede de sobreposição utilizada pela DHT. As DHTs de salto único OneHop [1] e D1HT [23] têm suas redes de sobreposição estruturadas em

anel, de forma similar à utilizada pelo Chord [15]. As estratégias de disseminação de informações na OneHop DHT e D1HT são descritas a seguir.

A disseminação de eventos na OneHop DHT é baseada em uma hierarquia em 3 níveis, onde cada nível é responsável por agrupar e desagrupar as informações sobre os eventos. Esta abordagem resulta em um alto nível de desbalanceamento de carga de mensagens para manutenção das tabelas de roteamento entre os nodos de diferentes níveis nesta hierarquia [23].

A D1HT utiliza um sistema de mensagem com tempo-de-vida para a disseminação dos eventos, onde o nodo que detecta um novo evento deve encaminhar esta informação para $log_2 N$ outros nodos. Antes de repassar adiante, os nodos agrupam os eventos que recebem para minimizar o uso da rede. Esta estratégia permite à D1HT disseminar a maioria dos eventos em tempo logarítmico, mas não garante que a latência máxima para a propagação de eventos também seja de ordem logarítmica.

Uma característica comum entre as DHTs de salto único propostas até o momento, é o fato de existirem situações que impossibilitam uma consulta ser respondida em um único salto, requerendo saltos adicionais para que a resposta seja obtida [29]. Em geral, esta situação é ocasionada quando um ou mais eventos estão em fase de disseminação na rede, período este no qual os nodos diferentes podem ter visões ligeiramente divergentes da configuração do sistema.

A necessidade de disseminar eventos entre os participantes de um sistema não é um requisito apenas das DHTs de salto único, outros sistemas computacionais, como os de diagnóstico distribuído, também precisam fazer uso de mecanismos para este fim. O algoritmo DiVHA *(Distributed Virtual Hypercube Algorithm)* e a rede HyperBone [22] são um exemplo de sistema de diagnóstico distribuído hierárquico eficiente e escalável, cuja latência máxima para disseminação de um evento qualquer é função logarítmica do número total de participantes.

A proposta do HyperDHT é fazer uso da infra-estrutura fornecida pelo algoritmo DiVHA para implementar os serviços necessários de uma DHT de salto único. O DiVHA

permite a cada *peer* calcular de forma independente seus enlaces virtuais, os quais interconectam os participantes em uma estrutura baseada em um hipercubo, formando a rede de sobreposição. Os enlaces virtuais são utilizados pelo HyperDHT para detecção e disseminação de eventos, e para troca das informações necessárias para manutenção da tabela de roteamento.

No topo desta infra-estrutura de base, o HyperDHT implementa as seguintes funcionalidades: mecanismos para posicionamento e busca dos objetos na rede P2P; particionamento e mapeamento consistente das chaves da tabela *hash* entre os *peers*; protocolos de entrada de novos participantes na rede; esquema de replicação das informações ou objetos armazenados na rede; e as funções DHT clássicas *lookup*, *put* e *get*.

O posicionamento e a posterior localização dos objetos distribuídos na rede é baseado em identificadores numéricos (chaves) que são atribuídos aos *peers* e aos objetos quando estes entram no sistema HyperDHT. O particionamento da tabela *hash* entre os *peers* é realizado através de técnicas de *hash* consistente [19]. Com isso, o remapeamento do espaço de chaves e a transferência dos objetos acontecem de forma localizada, somente entre *peers* adjacentes ao *peer* que originou um evento de entrada, saída ou falha.

Para ingressar em um sistema HyperDHT, um *peer* pretendente deve seguir o protocolo de entrada. Este procedimento determina, com base na topologia atual da rede, uma posição adequada para alocar o novo *peer*. A saída de um participante do sistema pode se dar de duas maneiras: o *peer* anuncia sua saída para seus vizinhos e segue o protocolo de saída para garantir a disponibilidade dos valores em um único salto; ou, ele simplesmente sai do sistema, ocasionando saltos extras para réplicas nas eventuais consultas em chaves pertencentes ao *peer* que saiu.

Os *peers* da HyperDHT mantêm uma tabela de roteamento completa, com referências para todos os demais participantes da rede. Cada entrada desta tabela associa o identificador de um determinado *peer* ao seu endereço de rede. A operação de consulta $lookup(key)$ faz uso desta tabela para encontrar o *peer* responsável por uma determinada chave e viabiliza a implementação das funções $put(key, value)$ e $get(key)$.

4.2 Modelo do Sistema

Considere um sistema S composto por um conjunto $\mathbb{V}$ de vértices, onde $v_i \in \mathbb{V}$ corresponde a um vértice do hipercubo $H(S)$, sendo que i determina o identificador numérico de v_i em $H(S)$. Os vértices deste sistema representam posições na rede de sobreposição que podem, ou não, ser ocupados por um *peer*. Se em um dado momento o vértice contém um *peer* ele é dito *ocupado*, caso contrário o vértice é dito *vazio*.

O sistema é considerado totalmente conectado, ou seja, os *peers* podem se comunicar diretamente, sem intermediários. Ao ingressar na rede, um *peer* recebe um identificador i que o relaciona a um dos vértices em $H(S)$. Um *peer* p_i qualquer pode estar em um de dois estados, *disponível* ou *indisponível*. O *peer* é dito *disponível* quando ele está em perfeitas condições operacionais, ou seja, interagindo da maneira esperada com os demais participantes do sistema. Um *peer indisponível* é aquele que, por alguma razão, apresenta um comportamento diferente do esperado. De forma geral, a indisponibilidade de um *peer* se dá pela ausência de comunicação dele com seus vizinhos na rede de sobreposição. O HyperDHT assume que as informações armazenadas nos *peers* são imutáveis. Não fazem parte dos objetivos deste trabalho as questões relativas a *peers* maliciosos, que poderão ser consideradas em trabalhos futuros.

Um evento é definido pela mudança de estado tanto de um vértice (*vazio*/*ocupado*) quanto de um *peer* (*disponível*/*indisponível*). O sistema é considerado dinâmico de forma que os *peers* podem entrar e sair da rede aleatoriamente, bem como os vértices e *peers* podem apresentar alternância de seus estados frequentemente. Um *peer* que passa do estado *indisponível* para *disponível* pode manter as informações que ele armazenava anteriormente, mas deve se atualizar sobre as eventuais mudanças ocorridas na configuração da rede antes de voltar à prover seus serviços.

Os *peers* são capazes de realizar testes em outros *peers* e determinar corretamente o estado do *peer* testado. Os testes são realizados periodicamente, dentro de um intervalo de testes fixo determinado em função dos requisitos desejados para o sistema. Um teste consiste em um procedimento executado pelo *peer* testador sobre o *peer* testado. A resposta

ou a falta da resposta dentro do intervalo esperado permite ao *peer* testador classificar o estado do *peer* testado entre *disponível* ou *indisponível*.

Uma rodada de testes é definida como o período de tempo em que todos os *peers* no estado *disponível* executam seus testes. A latência do sistema é definida como o tempo necessário, em rodadas de testes, para que todos os *peers disponíveis* descubram a ocorrência de um novo evento.

O grafo $H(S)$ é usado como base para construção da rede de sobreposição, a qual é representada pelo grafo $T(S)$. Em essência, $T(S)$ é uma cópia de $H(S)$ no qual, se necessário, são adicionadas arestas extras para compensar os caminhos interrompidos por vértices *vazios* ou *peers indisponíveis*. Uma aresta direcionada em $T(S)$ do *peer* i para o *peer* j representa o enlace virtual entre estes *peers*. Através dos enlaces virtuais são trocadas as mensagens de testes.

O grafo $T(S)$ é gerado de forma independente por cada *peer* através do algoritmo DiVHA. A figura 4.1 ilustra um grafo $T(S)$ para um sistema de 8 vértices *ocupados*. Os vértices brancos representam *peers disponíveis*, enquanto que os vértices em preto representam *peers indisponíveis*. O DiVHA utiliza $H(S)$ e o conhecimento dos resultados dos testes compartilhados entre os *peers* para garantir que o maior caminho entre qualquer par de *vértices* em $T(S)$ seja de no máximo $log_2 N$ arestas.

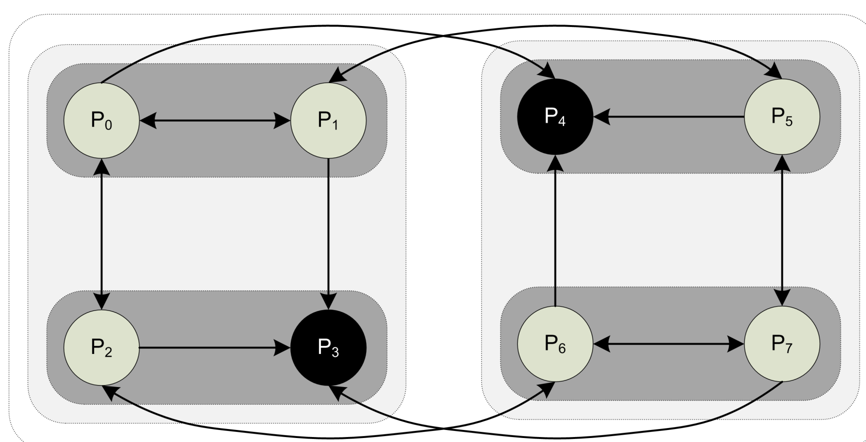

Figura 4.1: Grafo $T(S)$ para um sistema de 8 vértices *ocupados*.

A cada rodada de testes os *peers disponíveis* testam os *peers* adjacentes a ele em $T(S)$, este conjunto é denotado por $\mathbb{T}(i)$, sendo i o índice do *peer* testador. Para atualizar as informações sobre o estado dos participantes do sistema, um *peer* i ao testar um *peer* $p_j \in \mathbb{T}(i)$, p_i recebe de p_j informações sobre todos os *peers* k com distância de p_j menor ou igual a $log_2 N - 1$. A distância é definida como sendo o menor caminho em $T(S)$ entre um par de *peers* qualquer. Quando um *peer* p_i detecta um novo evento, ele executa o algoritmo DiVHA para atualizar $\mathbb{T}(i)$.

O valor máximo para a latência ($log_2 N$ rodadas de testes) e o valor máximo para a quantidade de testes necessários ($N \times log_2 N$ por rodada de testes) são garantidos pelo algoritmo DiVHA e são formalmente provados em [22].

4.3 Particionamento do Espaço de Chaves *Hash*

No HyperDHT as chaves são calculadas através da função criptográfica SHA-1 de 160bits, formando um conjunto de 2^{160} possíveis chaves. Subconjuntos deste espaço de chaves são associados aos vértices $v_i \in H(S)$. Cada vértice do hipercubo $H(S)$ recebe uma faixa de chaves determinada por $[i * 2^{160-d}, ..., (i+1) * 2^{160-d})$, onde i é o índice do vértice em $H(S)$, e d é a dimensão do hipercubo que modela o sistema HyperDHT corrente, que também corresponde à quantidade de níveis de *clusters* de $H(S)$, conforme mostra a figura 4.2.

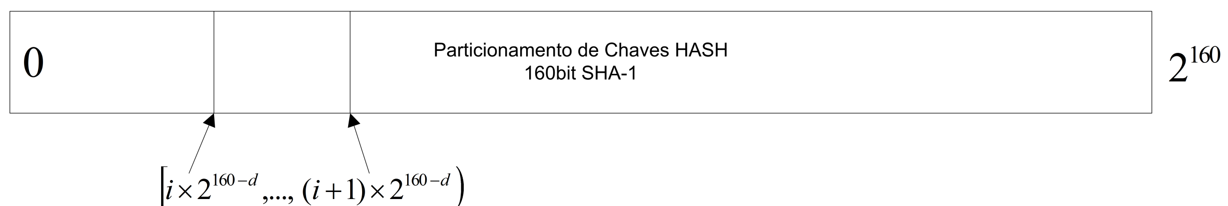

Figura 4.2: Representação gráfica do particionamento do espaço de chaves em um sistema HyperDHT.

Em um sistema completo, no qual todos os vértices de $H(S)$ estão sendo ocupados por *peers*, o mapeamento do espaço de chaves para cada *peer* p_i se dá de forma direta $1:1$, pelo índice i do vértice e do *peer*, ou seja, o *peer* p_i responderia pelo espaço de chaves associado ao vértice v_i. Nas configurações em que há a ocorrência de vértices *vazios*, o

peer cujo vértice é o mais próximo do vértice vazio em $H(S)$ fica responsável por aquele subconjunto de chaves. Este procedimento é detalhado a seguir.

Para encontrar o vértice mais próximo, o HyperDHT faz uso da seguinte propriedade. Cada vértice de um hipercubo é identificado por um número binário de n bits, no qual cada bit representa uma dimensão e cada dimensão tem dois valores possíveis. Dados dois identificadores de vértices, o número de bits que os diferem corresponde à menor distância entre estes vértices, e a posição do bit mais significativo que os difere determina a menor dimensão na qual estes vértices estão unidos. Para exemplificar, considere os vértices 2 (010) e 4 (100) da figura 4.3, eles estão separados por uma distância de dois passos, pois existem 2 bits que os diferem. Ainda neste exemplo, temos que o bit diferente mais significativo é o terceiro, logo a menor dimensão que contém ambos os vértices é $d = 3$.

Os vértices são agrupados em *clusters*, sendo a quantidade de vértices de um *cluster* definida por sua dimensão d e determinada por uma potência de 2 (2^d). Um mesmo vértice pode pertencer à *clusters* progressivamente maiores. A figura 4.3 ilustra como esse agrupamento é realizado para um sistema de 8 vértices. A lista dos vértices de um determinado *cluster* é calculada pela função $C(i, d)$, onde i é o índice de um dos vértices contidos no *cluster* de interesse, e d é a dimensão do *cluster*. A função $C(i, d)$ pode ser definida da seguinte forma: $C_{i,d} = i \oplus z, \forall z \in \mathbb{Z}_{\geq 0} : z \leq 2^d - 1$

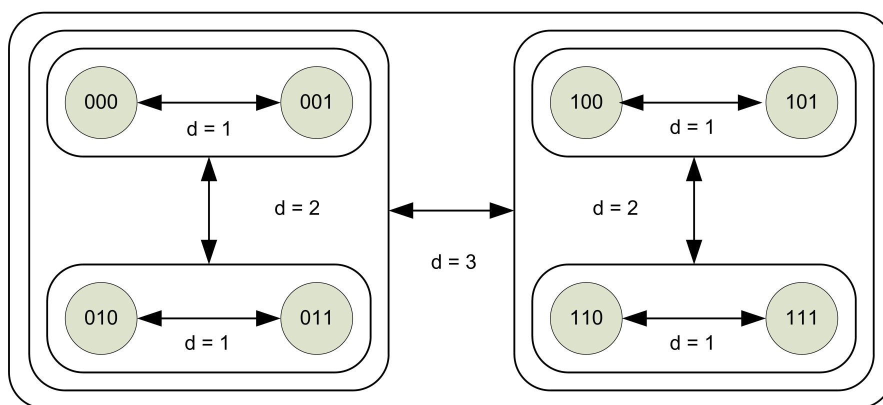

Figura 4.3: Identificadores binários e cálculo da distância entre vértices em um hipercubo.

Neste esquema de particionamento, um *peer* p_i responde pelo subconjunto das chaves associadas ao seu próprio vértice, somado aos subconjuntos das chaves dos vértices vazios

cuja diferença do identificador binário os torna mais próximos de p_i do que de qualquer outro *peer*, não importando o estado de p_i (*disponível* ou *indisponível*). Com isso, o HyperDHT consegue realizar *hash* consistente, onde os eventuais remapeamentos de chaves ocorrem somente nos *peers* adjacentes ao *peer* que originou um evento e os demais *peers* do sistema não são afetados.

O cálculo para determinar o *peer* responsável por um dado subconjunto de chaves é realizado de forma independente por cada um dos participantes do sistema. Este conhecimento é baseado nas informações do estado dos vértices do sistema. O algoritmo da figura 4.4 descreve como este cálculo é realizado.

```
01. find_vertice_owner ( v, S ) {
02.    z = 0
03.    i = v
04.    WHILE ( vertice i is empty in S )  {
05.        z++
06.        i = v XOR z
07.    }
08.    RETURN i
09. }
```

Figura 4.4: Pseudo-código - *peer* responsável pelas chaves associadas a um vértice.

A função `find_vertice_owner` recebe como parâmetro de entrada um identificador v, especificando o vértice para o qual se deseja encontrar o *peer* responsável, baseado no estado dos vértices que compõem o sistema S. Nas linhas 2 e 3 do pseudo-código da figura 4.4 são inicializadas duas variáveis auxiliares, z e i. Na quarta linha, temos um laço que é executado até que seja encontrado um vértice i em S cujo estado é *ocupado*. Na linha 5, z tem seu valor incrementado e este valor representa a distância entre dois vértices. O próximo vértice a ser testado pela condição do laço é calculado na linha 6, através da operação *ou-exclusivo* entre o identificador v e a distância z. O resultado deste algoritmo é consistente entre todos os participantes, mesmo nos momentos em que um ou mais parâmetros que descrevem o sistema estejam divergentes entre os *peers* devido a latência de propagação de eventos.

Para exemplificar, suponha que existe apenas um *peer* i ocupando a posição 0 no sistema da figura 4.5 e os demais vértices estão vazios. Nesta condição este *peer* fica responsável por todos os subconjuntos de chaves dos vértices deste sistema {0, 1, 2, 3}, pois o resultado da função `find_vertice_owner` para cada um dos vértices é 0, conforme exibido na tabela 4.1.

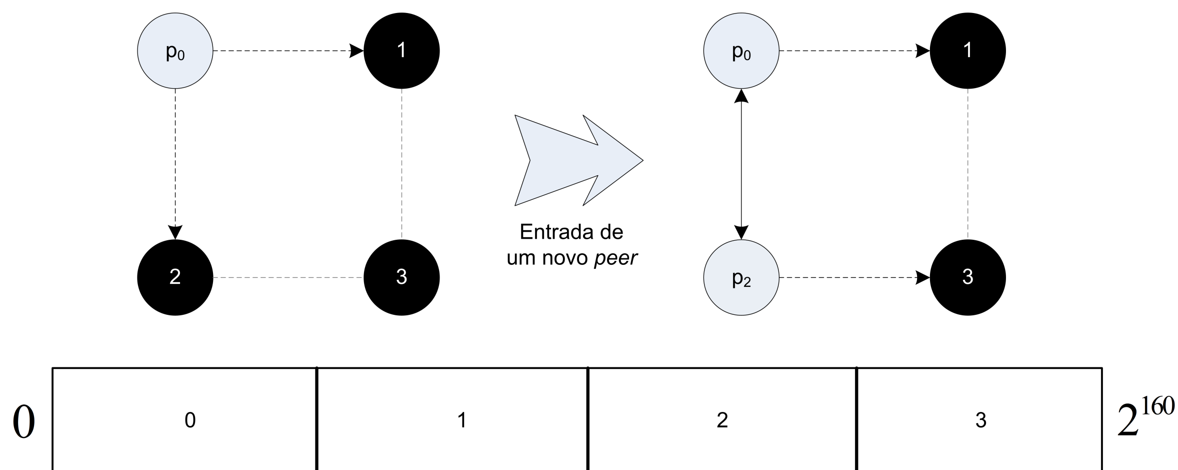

Figura 4.5: Divisão do espaço de chaves na entrada de um novo *peer*.

find_vertice_owner(0, S) => 0
find_vertice_owner(1, S) => 0
find_vertice_owner(2, S) => 0
find_vertice_owner(3, S) => 0

Tabela 4.1: Exemplo 1 - Resultados da função find_vertice_owner.

Considere que um novo *peer* p_j entre neste sistema e, pelas regras de entrada, p_j ocupa a posição 2. Nesta nova configuração o espaço de chaves é dividido entre os dois *peers*, onde p_i fica responsável pelas chaves dos vértices {0, 1}, e p_j pelas chaves dos vértices {2, 3}, conforme mostra a tabela 4.2.

find_vertice_owner(0, S) => 0
find_vertice_owner(1, S) => 0
find_vertice_owner(2, S) => 2
find_vertice_owner(3, S) => 2

Tabela 4.2: Exemplo 2 - Resultados da função find_vertice_owner.

A figura 4.6 estende o exemplo de particionamento do espaço de chaves para um sistema de 4 dimensões e 16 vértices, no qual inicialmente existem 6 *peers disponíveis*. Os *peers* estão ocupando os vértices em branco, enquanto que os vértices em preto estão *vazios*.

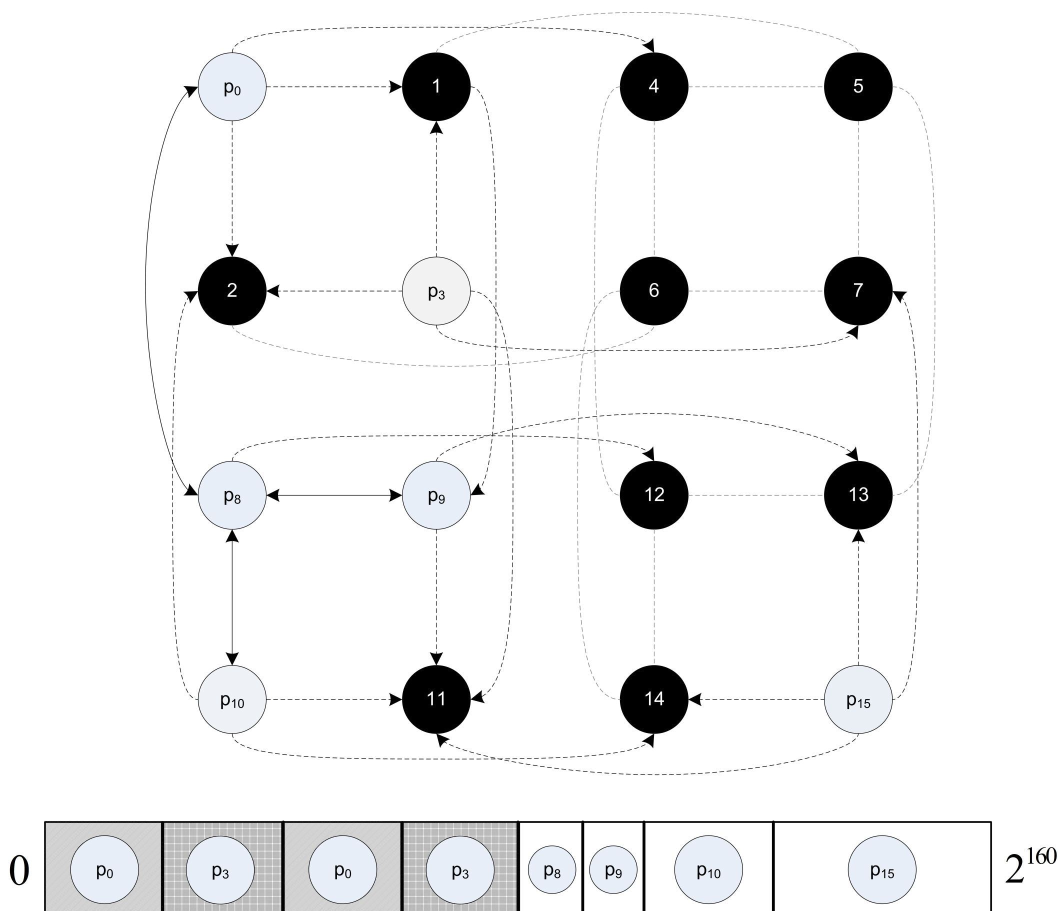

Figura 4.6: Exemplo de particionamento do espaço de chaves em um sistema HyperDHT.

Observe que neste exemplo de sistema a distribuição das chaves entre os *peers* não está balanceada, configurações como esta surgem devido à saída de *peer* de uma dada região. Entretanto, no HyperDHT isso é resolvido com a entrada de novos participantes, pois o protocolo de entrada irá posicionar os eventuais novos *peers* nas regiões menos povoadas. A figura também ilustra o particionamento do espaço de chaves, que está dividido em regiões e cada região contém o índice do *peer* responsável por aquela faixa de chaves. Note que, apesar da distribuição irregular dos *peers* no sistema, a distribuição do espaço de chaves é, relativamente, mais balanceada.

Para finalizar a explicação, suponha que o *peer* 8 saiu da rede, neste caso o *peer* 9 irá passar a responder pelo subconjunto de chaves deixados por p_8. Considere ainda que, algum tempo depois, p_9 e p_{10} saíram simultaneamente da rede, nesta situação o *peer* 15 será responsável por todas as chaves do *cluster* $C_{kd}(15, 3)$, o qual contém os vértices $[v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}]$.

4.4 Consultas em um Único Salto

O HyperDHT, a fim de viabilizar que cada consulta (operação de *lookup*) seja resolvida em um único salto, mantém uma tabela de roteamento com referências para todos os participantes do sistema. A taxa de sucesso real de uma consulta depende da precisão desta informação.

Neste trabalho denominamos o conhecimento sobre o estado de todos os participantes de um sistema distribuído como sua composição. Uma alteração na composição do sistema, que se dá na ocorrência de qualquer evento, levanta duas questões importantes. Primeiro, como um participante irá detectar o novo evento para atualizar sua visão local da composição e, segundo, como se dará a transmissão da informação sobre esta mudança para os demais *peers* da rede P2P, de modo que todos os participantes possam manter a informação atualizada sobre a composição e, consequentemente, realizar consultas em um único salto.

A primeira questão, referente à detecção de novos eventos por *peers* participantes do sistema, é resolvida no HyperDHT pelos protocolos de entrada e saída de *peers*, e pelo resultado dos testes que cada participante realiza baseado no grafo $T(S)$. O grafo $T(S)$ também auxilia o HyperDHT na solução da segunda questão, pois ele determina os enlaces virtuais que cada *peer* deve estabelecer e utilizar para troca de mensagens sobre as alterações na composição.

Para manter a visão da composição e a tabela de roteamento corretas e atualizadas, as notificações de eventos devem alcançar todos os participantes do sistema o mais rápido possível, mas sem comprometer os recursos de rede. O algoritmo DiVHA permite alcançar

este objetivo, pois ele fornece a garantia de que o maior caminho que uma notificação deverá percorrer para atingir todos os *peers* do sistema é de $log_2 N$ saltos, onde N é a quantidade de vértices existentes no sistema. Resultados da simulação mostram que, na média, a maioria dos eventos é totalmente propagada na metade deste tempo.

O processamento de uma consulta que recebe como entrada uma chave qualquer do espaço de chaves, implica em determinar corretamente qual o *peer* responsável pela chave. No HyperDHT as consultas são trivialmente resolvidas com base apenas no conhecimento local, o que torna possível realizar operações de *put* e *get* em apenas um único salto. O pseudo-código da figura 4.7 especifica como o algoritmo de *lookup* é executado no HyperDHT. O primeiro passo é determinar o vértice que acomoda a chave k, calculado pela expressão da linha 2. Em seguida, na linha 3, a chamada para função `find_vertice_owner` retorna o identificador do *peer* responsável pelo vértice v. Na quarta linha, este identificador é utilizado para indexar a tabela de roteamento que contém as informações necessárias para contactar o *peer* i.

```
01. lookup ( k, d, S ) {
02.    v = floor( k / pow( 2, 160-d ) )
03.    i = find_vertice_owner ( v, S )
04.    RETURN address of peer i from routing table
05. }
```

Figura 4.7: Pseudo-código - algoritmo *lookup* do HyperDHT.

No HyperDHT existe uma única situação onde podem ser necessários saltos adicionais para completar as operações *put* e *get*. Esta situação ocorre quando um *lookup* é realizado em uma chave cujo *peer* responsável tornou-se *indisponível*, e a notificação associada a este evento ainda está em fase de propagação. Neste caso, não é possível concluir com sucesso a operação e, com base nesta resposta, o participante que gerou a consulta atualiza sua visão da composição do sistema e determina o *peer* que passou a responder por aquela chave, sem precisar esperar pela notificação que ainda está sendo disseminada. A disponibilidade do valor associado à chave é garantida pelo esquema de replicação, assunto da próxima seção.

4.5 Esquema de Replicação

Para garantir que as informações ou os objetos armazenados em uma DHT mantenham a disponibilidade mesmo quando o *peer* encarregado pelo armazenamento não tem disponibilidade 100% garantida requer, inevitavelmente, alguma forma de redundância. O HyperDHT implementa um esquema de replicação, onde cada par chave/valor associado a um *peer* pode, opcionalmente, ser replicado em k outros *peers* do sistema. A lista ordenada dos k *peers* que recebem uma dada réplica é chamada de "cadeia de replicação". O valor do parâmetro k pode ser ajustado conforme a aplicação, de acordo com o grau desejado de disponibilidade das informações. Redes P2P maiores e com alta ocorrência de *churn* necessitam de um grau de replicação maior, se comparado com sistemas menores e mais estáveis.

Para fins de replicação, o HyperDHT considera que informações armazenadas nos *peers* são imutáveis e, portanto, não é preciso realizar procedimentos de atualização e sincronia entre os valores originais e suas réplicas. Desta forma, operações de *get* em réplicas retornam o valor correto e esperado.

Para um sistema onde o valor de k é maior que 0, o *peer* que recebe um novo valor, através de uma operação *put*, deve iniciar a geração das k réplicas exigidas. A escolha dos *peers* que receberão as réplicas é definida pelo pseudo-código da figura 4.8. A função `make_replicas` segue a mesma estratégia usada pela função `find_vertice_owner`, como consequência o participante que assume o conjunto de chaves de um *peer* que saiu do sistema, ou tornou-se *indisponível*, possui uma réplica dos valores e poderá, de imediato, responder por aquelas chaves.

O ingresso de novos *peers* em um sistema com replicação exige que seja verificado se este novo *peer* faz parte da cadeia de replicação do participante que o aceitou, caso positivo, ele deve receber uma réplica dos valores armazenados pelo *peer* que dividiu seu subconjunto de chaves. Esta verificação também deve ser realizada pelos demais participantes ao receber a notificação sobre a entrada do novo *peer*.

```
01. make_replicas ( v, S, k ) {
02.    z = 0
03.    i = v
04.    WHILE ( k > 0 )  {
05.        z++
06.        i = v XOR z
07.        IF ( vertice i is available in S ) {
08.            make replica of values of v in i
09.            k--
10.        }
11.    }
12. }
```

Figura 4.8: Pseudo-código - replicação de valores em um sistema HyperDHT.

De forma similar, deve-se verificar se a quantidade de réplicas k não foi comprometida pela saída de *peers* e, caso necessário, o atual responsável pelos valores deve executar a função `make_replicas` para compensar as réplicas que deixaram de existir. Esta verificação deve acontecer sempre quando um *peer* receber a notificação de saída de um participante. Eventualmente, podem existir no sistema mais do que k réplicas de um dado valor. O conhecimento local dos participantes permite, caso se julgue necessário, identificar e descartar as réplicas extras.

4.6 Protocolo de Entrada em uma Rede HyperDHT

A entrada de um *peer* em uma DHT envolve o problema do posicionamento deste novo participante na topologia do sistema. É prática usual dos sistemas DHT [15, 33, 37, 23, 1], deixar esta questão por conta da função *hash* e acreditar que o efeito avalanche irá fazer uma distribuição razoavelmente homogênea dos participantes na rede de sobreposição. O HyperDHT utiliza uma abordagem alternativa ao incorporar no protocolo de entrada (1) mecanismos para posicionar deterministicamente o novo participante no local da rede onde ele é mais necessário, ou seja, onde o subconjunto do espaço de chaves é o maior associado a um único *peer*; (2) procedimentos para garantir que as chaves e valores assinalados para o novo *peer* continuem disponíveis em um único salto durante o período de propagação deste evento.

Um sistema HyperDHT é inicializado com tamanho $d = 1$ e o *peer* que cria um novo sistema assume o identificador (id) 0 em $T(S)$. A figura 4.9 é uma representação gráfica de um sistema HyperDHT recém criado e inicializado, note que a quantidade de vértices neste hipercubo de dimensão $d = 1$ é de $N = 2^1 = 2$. O vértice de linha contínua foi ocupado pelo *peer* que criou este sistema, e o vértice tracejado está vazio e pode acomodar um novo participante que pretenda ingressar nesta rede.

Figura 4.9: Uma rede HyperDHT recém criada e inicializada.

Para ingressar em um sistema HyperDHT existente, o *peer* pretendente deverá conhecer pelo menos um participante desta rede. Suponha que p_n tem conhecimento sobre o *peer* p_2 e está interessado em entrar na rede ilustrada pela figura 4.10.

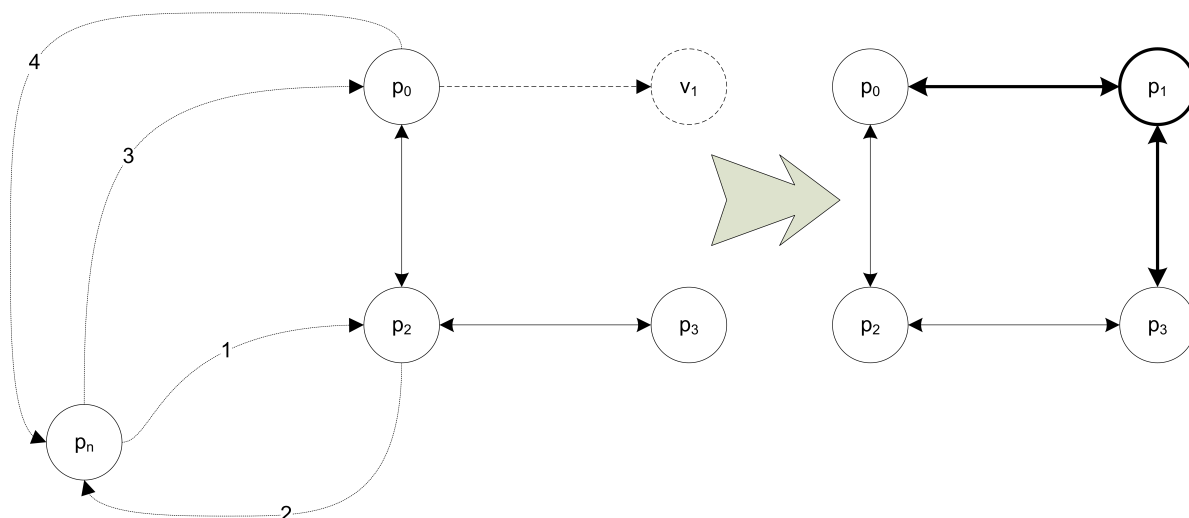

Figura 4.10: Procedimento de entrada de um novo *peer* em uma rede HyperDHT.

O primeiro passo realizado por p_n é requisitar para p_2 (Passo 1) um identificador de vértice e endereço de rede do *peer* que ele deve contactar para dividir o espaço de chaves. Ao receber esta requisição, p_2, baseado apenas em sua visão da composição do sistema, faz uma busca pelo *peer* cujo subconjunto de chaves é o maior em relação aos demais. Se, durante esta busca, p_2 constatar que não existe nenhum vértice *vazio*, ele deve realizar o procedimento de aumento do sistema para que novos vértices sejam criados.

Aumentar o sistema corresponde a incrementar em uma unidade a dimensão de $H(S)$, o que implica em dobrar o número total de vértices da rede HyperDHT. Este procedimento também envolve um ajuste das posições dos *peers* em $T(S)$, para que o mapeamento das chaves *hash*, em relação aos *peers*, se mantenha inalterado. O pseudo-código da figura 4.11 mostra como este aumento é executado. Na linha 2 o parâmetro que indica a atual dimensão de S é incrementado e, no laço que segue, o identificador de todos os *peers* tem seu valor multiplicado por 2.

```
01. system_expand ( S ) {
02.     increment d of S in one unit
03.     FOREACH ( peer in S AS p ) {
04.         p.id = p.id << 1
05.     }
06. }
```

Figura 4.11: Pseudo-código - algoritmo de incremento do sistema em uma dimensão.

É importante lembrar que este cálculo é resolvido localmente por cada um dos *peers*, ou seja, o aumento no tamanho do sistema não implica na disseminação dos novos índices de *peers* pela rede, basta disseminar a nova dimensão d. A figura 4.12 mostra um sistema HyperDHT antes e depois da execução do algoritmo `system_expand`.

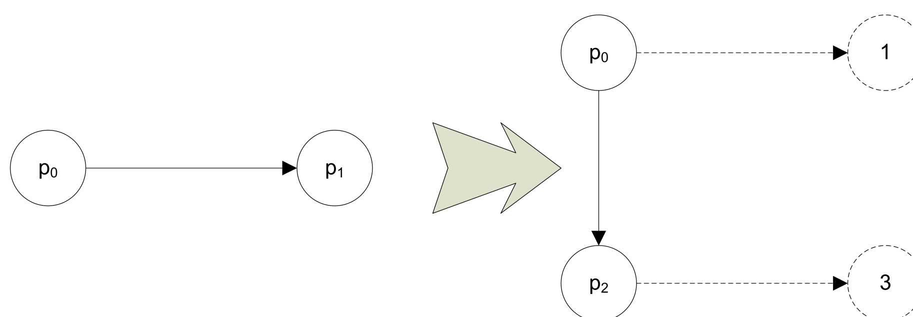

Figura 4.12: Expansão de um sistema HyperDHT.

Note que, mesmo durante o período de propagação do novo d, onde *peers* diversos podem ter visões diferentes do tamanho do sistema, as operações da DHT não são afetadas, pois o mapeamento das chaves independe da noção de tamanho do sistema HyperDHT.

Continuando com o exemplo da figura 4.10, na existência de vértices *vazios* no sistema, p_2 consegue finalizar sua busca pelo vértice mais adequado para o novo participante. Esta informação é repassada para p_n juntamente com o endereço do *peer* p_0 (Passo 2), sendo este o atual responsável pelo subconjunto de chaves do vértice em questão. Então p_n contacta p_0 (Passo 3) para confirmar se é possível sua entrada no vértice indicado. A confirmação é importante pois podem ocorrer múltiplas requisições de entrada direcionadas para um mesmo *peer* e, portanto, é necessário que este *peer* gerencie a entrada de novos participantes para evitar que mais de um participante ocupe um mesmo vértice.

No caso em que o *peer* negue a entrada de um novo participante em um determinado vértice, ele deve fornecer ao pretendente um novo identificador de vértice e o endereço do *peer* relacionado para que o *peer* pretendente possa realizar uma nova tentativa de entrada. Caso contrário, p_0 informa a p_n (Passo 4) que ele foi aceito na rede e fornece uma cópia de seu conhecimento do sistema, que engloba a dimensão atual de $H(S)$, sua visão local da composição e a tabela de roteamento.

A ilustração mais a direita da figura 4.10 mostra o sistema com p_n já ocupando a posição do vértice 1 no hipercubo. A partir deste ponto, este vértice torna-se *ocupado* e p_0 negará as subsequentes requisições de entrada para este vértice. Por fim, p_n, agora p_1, deve resgatar todos os subconjuntos de chaves/valores dos vértices dos quais ele é o responsável. Parte deste subconjunto vem do *peer* que o aceitou na rede, mas pode ser necessário contactar outros *peers* para obter todos os valores das chaves associadas a ele.

Por consequência das regras de replicação, p_0 naturalmente fará parte da cadeia de replicação de p_1, logo, a disponibilidade dos valores transferidos para p_1 é garantida mesmo durante o período de propagação deste evento de entrada, pois p_0 é capaz de responder por estas consultas. Se p_0 receber operações de *put* durante o período de propagação, ele deve atualizar p_1 com o novo valor antes de confirmar o sucesso da operação para o participante que a originou.

4.7 Saída de Participantes do HyperDHT

O HyperDHT prevê duas formas de saída de um *peer* da rede. A primeira ocorre quando o *peer* anuncia sua saída para seus vizinhos. Neste caso, ele permanece no sistema por pelo menos d rodadas de testes antes de desconectar. Durante este período, ele ainda deve responder pelas requisições endereçadas para suas chaves e, se necessário, ele também deve realizar as devidas replicações. Somente após este tempo, o *peer* poderá efetivamente deixar a rede, com isso é garantido que as consultas continuem sendo respondidas em um único salto.

A segunda situação ocorre quando o *peer* simplesmente se desconecta do sistema HyperDHT sem aviso prévio, como por exemplo pela falta de energia ou falha de rede. Nesta situação este *peer* será considerado *indisponível* pelos seus testadores e, se esta condição persistir por mais de um determinado tempo, ele será automaticamente removido da rede. O esquema de replicação permite que a saída dos *peers* desta forma possa acontecer sem perda das informações. O participante que irá assumir as chaves é um dos que já mantém uma réplica dos valores armazenados no *peer* que saiu, e poderá passar a responder por estas chaves imediatamente após a exclusão do *peer indisponível*.

Durante o período em que um participante está em estado *indisponível*, as operações endereçadas ao subconjunto de chaves associadas a este *peer* devem ser redirecionadas a um dos *peers* que mantém réplica, seguindo a ordem da cadeia de replicação. Nesta situação, as operações nas réplicas são resolvidas em um único salto. Saltos extras são necessários apenas quando o evento sobre a indisponibilidade ainda está em fase de propagação e o *peer* requerente ainda não recebeu esta notificação.

Na condição do sistema HyperDHT não utilizar replicação, a indisponibilidade de um *peer* também ocasiona a indisponibilidade das informações armazenadas por ele. Neste caso, uma operação endereçada a este subconjunto de chaves falha, e é sinalizado ao *peer* requerente que não foi possível executar a requisição solicitada. A única opção neste caso é esperar até que o *peer* volte ao estado *disponível*, ou até que ele seja removido do sistema, e a consulta possa ser direcionada a outro *peer*.

Um *peer* que passa do estado *indisponível* para *disponível* pode manter as informações que ele armazenava anteriormente, mas deve se atualizar sobre as eventuais mudanças ocorridas na configuração do sistema. Bem como, no caso de existir replicação, deve resgatar os valores inseridos durante o período de indisponibilidade.

CAPÍTULO 5

RESULTADOS EXPERIMENTAIS

Neste capítulo são apresentados os resultados obtidos das simulações conduzidas com o sistema HyperDHT. A implementação foi realizada em linguagem de programação C e as simulações foram executadas utilizando a biblioteca de simulação de eventos discretos SMPL (*Simple Portable Simulation Language*) [24].

Foram executadas duas séries de simulações, apresentadas respectivamente nas seções 5.1 e 5.2. A subseção 5.1.1 apresenta uma análise da latência para os resultados da primeira série e a subseção 5.1.2 traz uma discussão sobre o consumo dos recursos de rede. A subseção 5.2.1 apresenta a análise dos efeitos de diferentes cargas de eventos em um sistema HyperDHT. E, por fim, a subseção 5.2.2 analisa o impacto do intervalo de testes no consumo dos recursos de rede.

5.1 Primeira Série de Simulações

A primeira série teve como objetivo mensurar a latência para propagação de eventos e o uso dos recursos de rede para manutenção do sistema HyperDHT. O sistema simulado teve seu tamanho progressivamente aumentado, sendo que a carga de eventos foi mantida equivalente em relação ao tamanho do sistema. Foram simulados sistemas de 1 até 9 dimensões, o que corresponde a tamanhos de rede de 2 até 512 vértices. Todos os vértices foram iniciados em estado *ocupado* e com o respectivo *peer* em estado *disponível*.

Durante a simulação, a produção de um novo evento foi condicionada à inexistência de eventos em fase de propagação, ou seja, um novo evento é criado apenas quando o evento anterior é completamente propagado. Os vértices geradores de eventos são escolhidos

de forma randômica, da mesma maneira que o evento, entre os quatro tipos possíveis, é escolhido aleatoriamente. No caso do vértice sorteado não suportar o evento escolhido, como por exemplo um evento de *indisponibilidade* em um vértice *vazio*, um novo sorteio é realizado até que se encontre uma combinação possível.

5.1.1 Latência e Tempo para Disseminação de Eventos

A latência para disseminação de um evento foi mensurada pelo número de rodadas de testes necessários para que todos os *peers* em estado *disponível* fossem notificados sobre a ocorrência do evento. A sumarização dos dados coletados é exposta na figura 5.1.

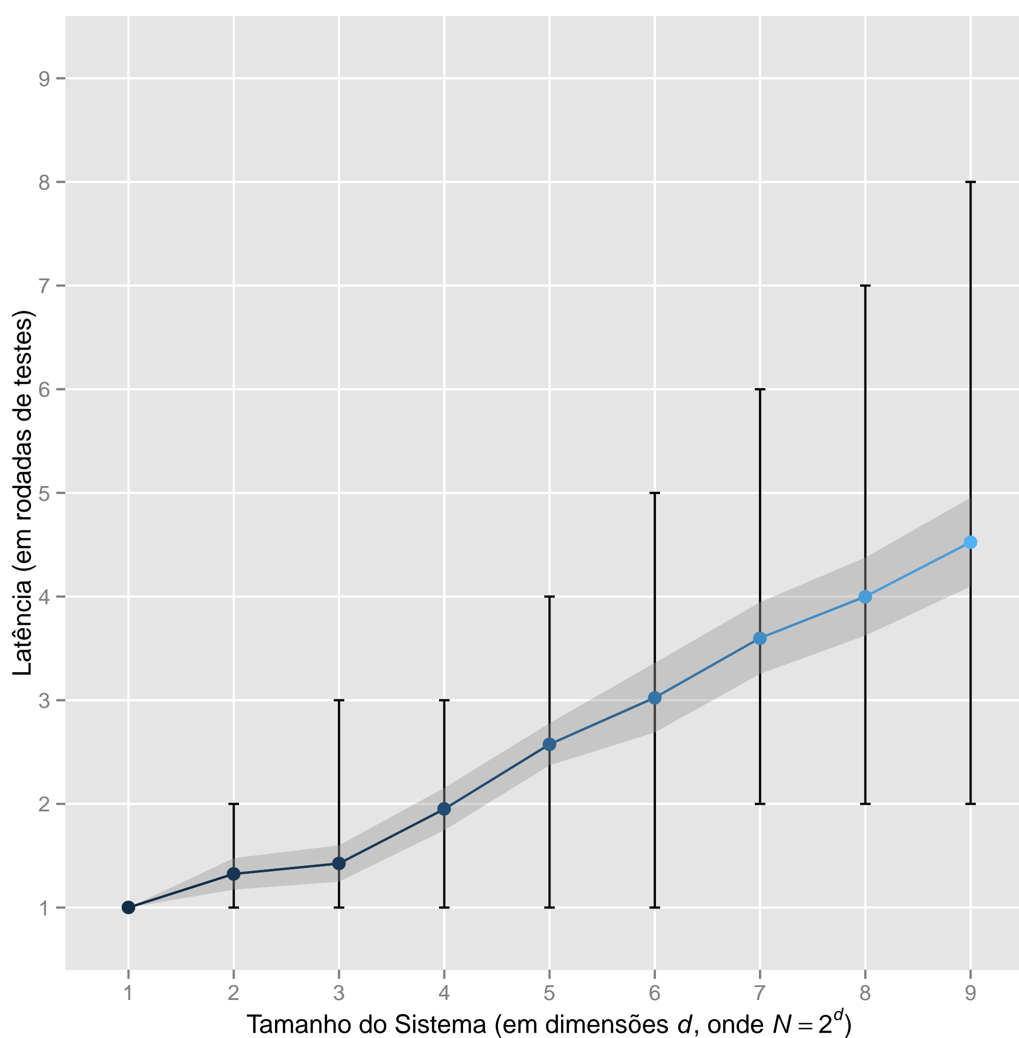

Figura 5.1: Latência média com intervalo de confiança de 95%.

A curva com pontos representa a latência média, em rodadas de testes, para cada tamanho de sistema simulado. A área mais escura ao redor desta curva corresponde ao intervalo de confiança de 95%, e as barras verticais mostram as latências máxima e mínima registradas.

O resultado da simulação demonstra que, na média, os eventos são completamente disseminados em um número de rodadas de testes equivalente à metade da latência máxima teórica ($log_2 N$). O intervalo de confiança mostra que a distribuição das latências está 95% concentrada próxima ao valor médio. Os valores máximos e mínimos registrados estão dentro do limite teórico especificado pelo algoritmo DiVHA.

Durante a simulação também foi registrado o tempo, em segundos, necessário para a propagação completa de um evento. Note que este valor depende diretamente do intervalo em que os *peers* realizam seus testes, sendo que nesta simulação o intervalo entre os testes foi de 30 segundos. O gráfico da figura 5.2 traça a curva do valor médio desta métrica em função do tamanho do sistema.

Em teoria, o tempo médio em segundos pode ser calculado multiplicando o valor médio da latência em rodadas de testes pelo intervalo de testes. Tomando como exemplo o sistema com dimensão $d = 8$ da figura 5.1, temos que a latência média é de 4 rodadas. Sendo o intervalo de testes de 30s, o tempo médio para propagação de um evento deveria ser de 120s ($4 * 30$). Entretanto, o resultado observado na prática, expresso na posição 256 do gráfico da figura 5.2, é de aproximadamente 150s. Esta diferença é esperada pois, na prática, existem outros fatores que influenciam no tempo, como o intervalo de tempo que existe entre a ocorrência de um evento até sua detecção.

A escolha do intervalo de testes de 30 segundos nesta simulação se deu pois é um intervalo tipicamente encontrado em trabalhos de diagnóstico distribuído. Caso se julgue necessário, sempre é possível utilizar intervalos de testes menores para diminuir o tempo de propagação dos eventos mas, nesta condição, o consumo dos recursos de rede será maior.

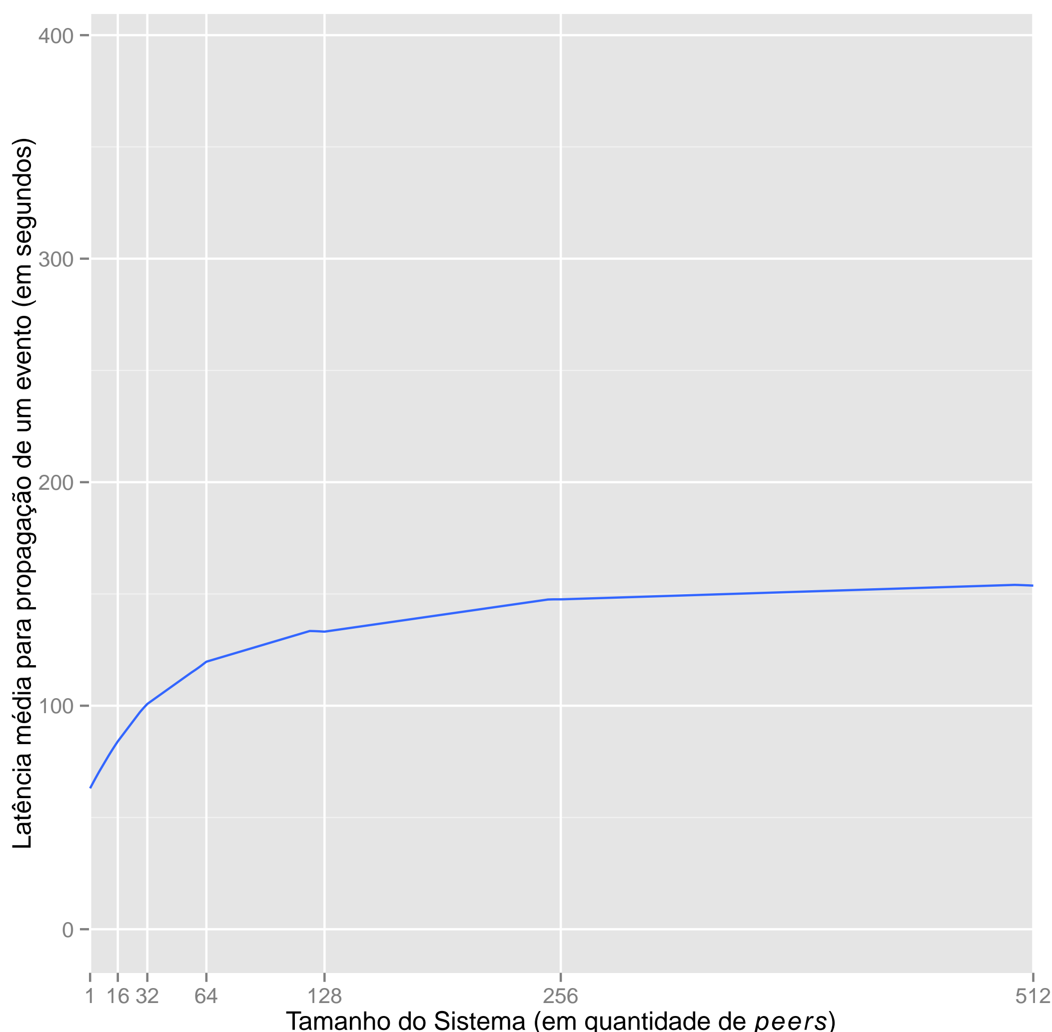

Figura 5.2: Latência média em segundos, para intervalo de testes de 30s.

5.1.2 Consumo dos Recursos de Rede

Para avaliar o custo de manutenção do HyperDHT também foram computados, durante a primeira série de simulações, a quantidade de mensagens e o volume de dados trocados entre os *peers*. O volume total de dados inclui o *overhead* decorrente dos cabeçalhos da pilha de protocolo TCP/IP.

O consumo da banda de rede total, acumulado de todos os *peers*, é exibido no gráfico da figura 5.3. O intervalo de testes é de 30 segundos e existe, no sistema simulado, a ocorrência de vértices *vazios* e/ou *peers indisponíveis*. Pelo conjunto de dados obtidos da simulação é possível observar que a taxa de crescimento do consumo da banda de rede

é ligeiramente maior em relação à taxa de crescimento do sistema. A razão para isso é o fato que, quanto maior a quantidade de *peers* no sistema, mais mensagens deverão ser geradas para uma mesma notificação de evento.

Analisando o maior sistema simulado com 512 *peers*, obtivemos um consumo de banda de rede total de aproximadamente 112 Kbps, o que é um custo bastante aceitável para manutenção da tabela de roteamento de um sistema deste tamanho. Se considerarmos a perspectiva do consumo médio da banda de rede individualmente por cada *peer*, fica mais claro que o custo a ser pago é baixo pelo benefício de realizar consultas em um único salto.

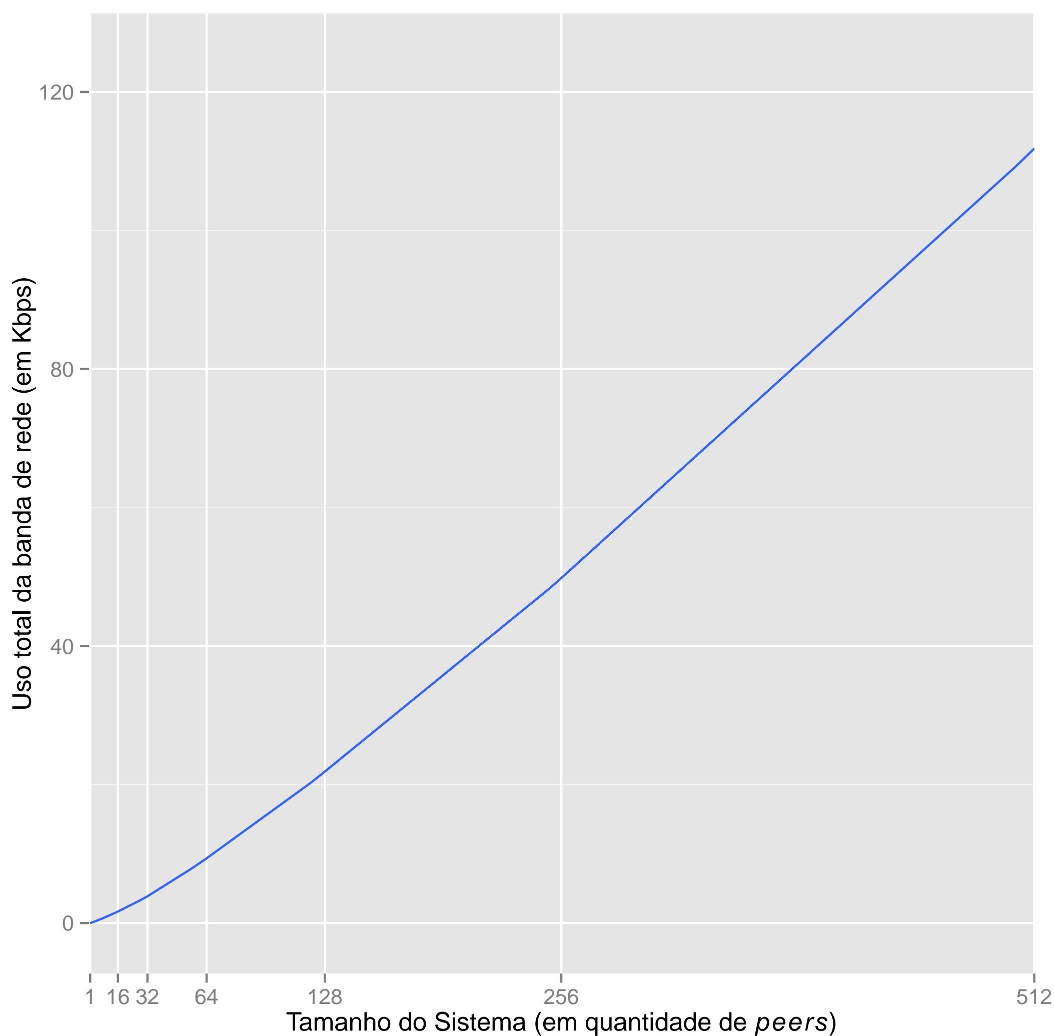

Figura 5.3: Uso total dos recursos de rede para manutenção do HyperDHT.

O próximo gráfico, da figura 5.4, mostra o uso médio dos recursos de rede por *peer*. Os valores aferidos são relativamente inexpressivos se comparados com a disponibilidade de banda atualmente oferecida. Com intervalo de testes de 30 segundos seria possível, inclusive, fazer uso de tecnologias mais antigas como os modens *dial-up* para conexão de *peers* em uma rede HyperDHT. Ou, diante da quantidade de banda de rede disponível, pode-se considerar o uso de intervalos de testes menores para redução da latência e, consequentemente, minimizar os casos onde saltos extras em consultas são necessários.

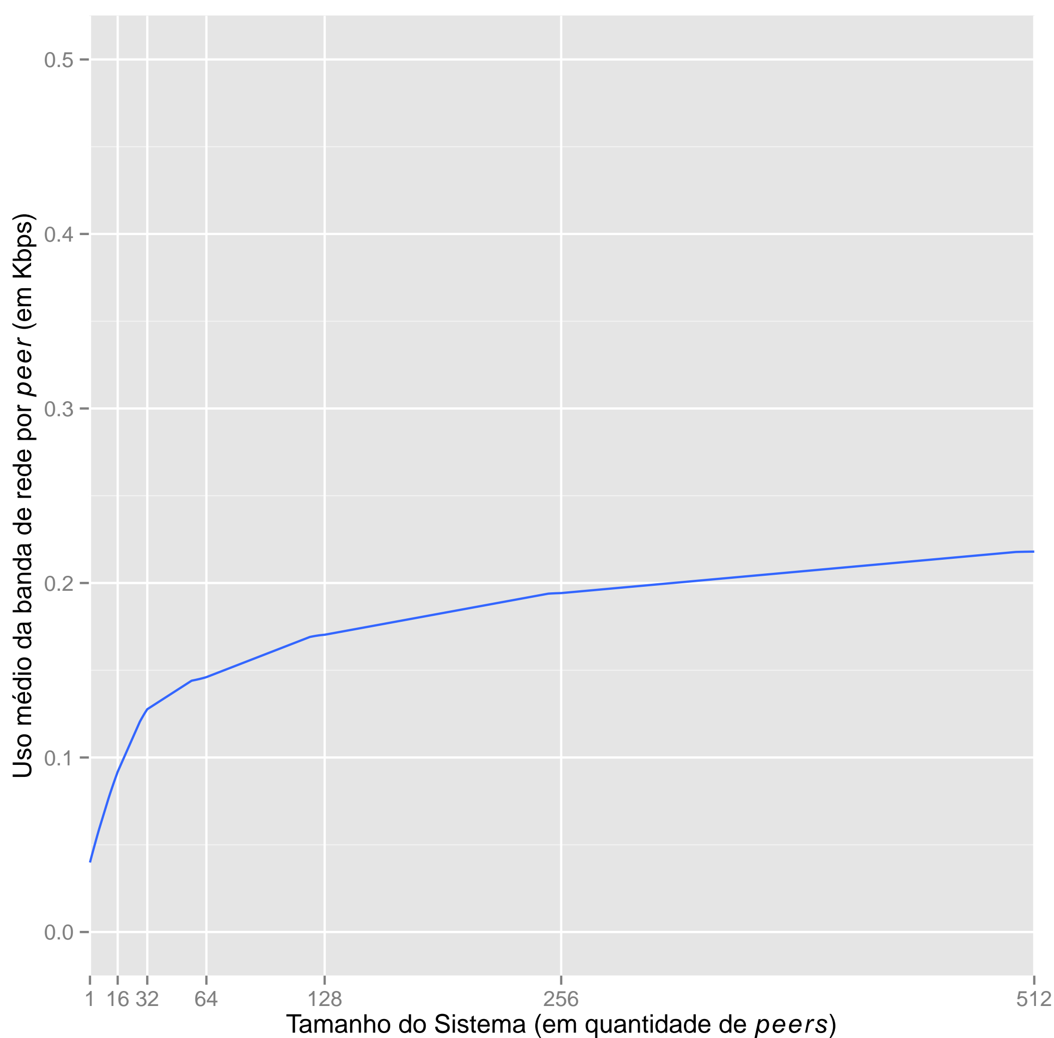

Figura 5.4: Uso médio dos recursos de rede por peer.

Por fim, cabe salientar que os resultados do consumo dos recursos de rede da D1HT e OneHop DHT são originalmente apresentados em outra métrica ou não consideram o *overhead*, impossibilitando a comparação com os resultados obtidos pelo HyperDHT.

5.2 Segunda Série de Simulações

A segunda série de simulações teve como objetivo principal verificar o impacto de diferentes cargas de eventos (*churn*) na taxa de sucesso das operações *get* em um único salto. O sistema simulado teve dimensão $d = 8$ de 256 vértices, onde cada vértice foi inicializado em estado *ocupado* com seus respectivos *peers* em estado *disponível*.

No primeiro instante da simulação foram introduzidos, através de operações *put*, valores pré-determinados na DHT, os quais posteriormente eram utilizados para testar o sucesso das operações *get* em um único salto. Os *gets* foram realizados ao final de cada rodada de testes, para tanto os *peers* sorteavam uma chave entre as previamente introduzidas. O fator de replicação foi de $k = 9$, ou seja, existiam no mínimo 10 cópias de um dado valor em *peers* distintos. Somente após a fase de inserção e replicação dos valores é que se iniciou a geração de eventos. O vértice e o tipo do evento, como na primeira série, foram escolhidos de forma aleatória, mas sem a condição que impedia eventos simultâneos. O sistema também não podia ter sua dimensão alterada, o que implicou ignorar entradas de novos *peers* no caso de todos os vértices estarem em estado *ocupado*.

No caso da ocorrência de um evento de saída de um *peer*, o simulador também deveria escolher entre os dois tipos de saída possíveis: (1) seguindo o protocolo de saída, ou (2) saída imediata, sem comunicar os vizinhos. Para este fim, foi determinado que a chance do *peer* sair de forma imediata é de 33%. Caso contrário, o *peer* permanece na rede por d rodadas de testes, honrando o protocolo de saída para garantir a disponibilidade dos valores em um único salto.

Para uma dada configuração de sistema, foram simuladas 3 cargas diferentes de eventos, onde a carga de eventos 1 indica que o simulador manteve apenas 1 evento concorrente por vez. Na carga de eventos 2 eram mantidos 2 eventos concorrentes durante a simulação e, da mesma forma, a carga de eventos 3 corresponde a 3 eventos concorrentes. Além disso, também foram simulados sistemas com diferentes intervalos de testes (30, 20 e 10 segundos), para verificação do efeito deste parâmetro no consumo dos recursos de rede. O resultado desta série é composto pelos dados obtidos em 9 simulações distintas.

5.2.1 Análise do *Churn* na Taxa de Sucesso das Consultas

Durante a simulação foram registradas a quantidade de *gets* respondidos em um único salto, bem como os *gets* que necessitaram de saltos extras para obtenção da resposta. A quantidade de *gets* realizados por simulação depende do intervalo de testes simulados, sendo que a menor quantidade de *gets* registrado em uma simulação foi de 29.130, e a maior foi de 67.496.

O gráfico da figura 5.5 sumariza, com foco na ocorrência de *gets* em múltiplos saltos, os resultados da segunda série de simulações. O valor das barras representa a porcentagem desta ocorrência em relação à quantidade total de *gets*. Os resultados são apresentados em função da carga de eventos e estão agrupados pelos respectivos intervalos de testes.

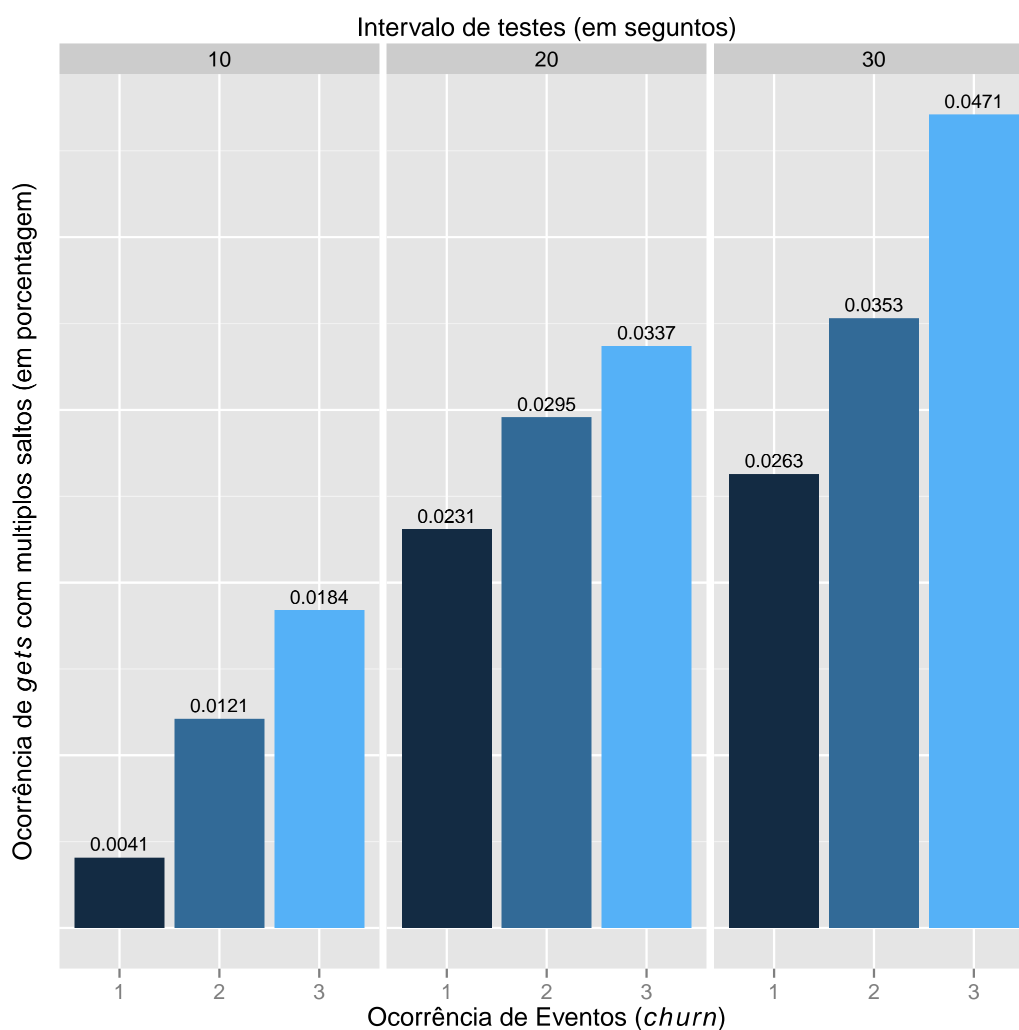

Figura 5.5: Impacto do *Churn* na taxa de sucesso de *gets* em salto único.

A configuração que apresentou a maior ocorrência de *gets* em múltiplos saltos (0.0471%) foi o sistema com intervalo de testes de 30 segundos e carga de eventos 3. Este resultado condiz com o que era esperado, pois quanto maior o intervalo entre os testes, potencialmente maior será o tempo para detecção de um evento e para sua propagação. Outro fator que tem influência na taxa de sucesso de *gets* de salto único é a dimensão do sistema pois, quanto maior a dimensão, potencialmente maior será o número de rodadas de testes necessário para a propagação de um evento. Em ambos os casos, maior será a probabilidade de incidência de *gets* em uma chave relacionada a um *peer* cujo evento de *indisponibilidade* está em fase de disseminação.

A quantidade de *gets* de múltiplos saltos registrada foi inexpressiva em relação à ocorrência total de *gets*, não impactando no funcionamento geral dos sistemas simulados. A D1HT e a OneHop DHT não apresentam um estudo relativo a este parâmetro, impossibilitando uma comparação. De qualquer forma, se considerarmos que as três DHTs têm, no pior caso, tempo de disseminação logarítmico em função do tamanho do sistema, o HyperDHT é o único que apresenta, nos protocolos de entrada e saída, mecanismos para aumentar a disponibilidade dos valores em um único salto. Isso é possível no HyperDHT pois o DiVHA garante a latência máxima. O mesmo não ocorre na D1HT e na OneHop DHT, onde a latência máxima, em certas circunstâncias, pode ser maior que o log_2 de N.

5.2.2 Intervalo de Testes e o Consumo dos Recursos de Rede

Durante a segunda série de simulações também foi monitorado o consumo dos recursos de rede. O gráfico da figura 5.6 apresenta estes resultados, sendo que o valor das barras mostra o consumo total da rede em Kbps, considerando os testes, mensagens e o *overhead* da pilha de protocolo TCP/IP. Os resultados estão agrupados pelos respectivos intervalos de testes e são apresentados em função da carga de eventos.

Através deste gráfico, é possível verificar que a carga de eventos teve pouca influência no consumo total dos recursos de rede, de forma contrária ao intervalo de testes que mostrou ter uma grande influência.

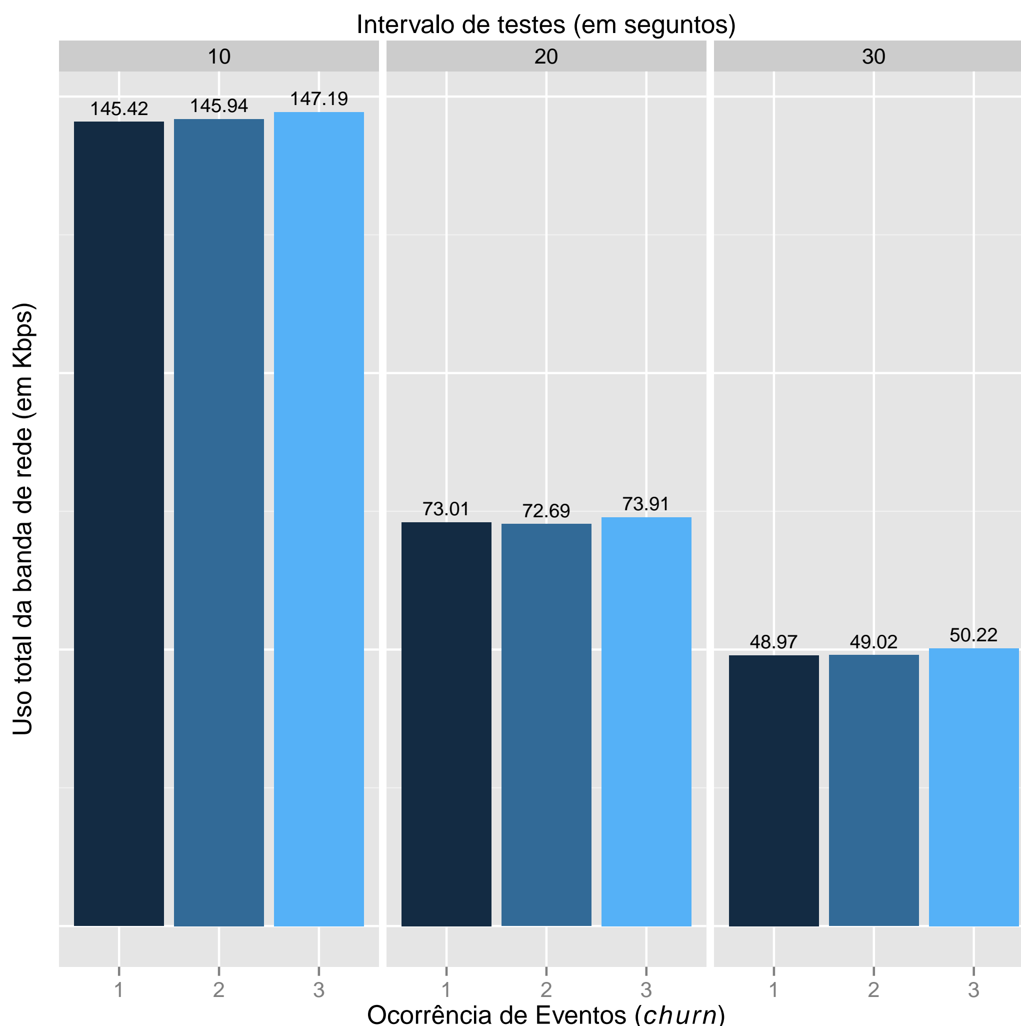

Figura 5.6: Análise do intervalo de testes no consumo da banda de rede.

Este resultado era previsto pois a maior fonte de tráfego de rede tem origem nas mensagens de testes, visto que estes ocorrem periodicamente em oposição às mensagens de diagnóstico que acontecem pontualmente. Desta maneira, espera-se que o consumo dos recursos de rede dobre na mesma razão em que o intervalo de testes diminua pela metade. O tamanho do sistema também impacta no consumo total dos recursos de rede. Entretanto, conforme visto no gráfico 5.4, o consumo específico por cada *peer* cresce de forma logarítmica em função do tamanho do sistema. O que atribui ao HyperDHT qualidades que auxiliam na escalabilidade, visto que geralmente a preocupação em relação ao consumo de rede está relacionada aos *links* que conectam os *peers* na rede, e não ao consumo total do sistema.

CAPÍTULO 6

CONCLUSÕES

As DHTs são estruturas de dados distribuídas que associam um identificador único aos dados para possibilitar a indexação, distribuição e localização de forma descentralizada e escalável. As primeiras DHTs introduzidas são do tipo *multi-hop* pois necessitam que uma consulta seja roteada por diversos nodos antes de obter a resposta. As *single-hop* DHTs se diferenciam das *multi-hop* por permitirem que as consultas sejam respondidas em um único salto. No entanto, as *single-hop* DHTs necessitam que cada *peer* mantenha uma tabela de roteamento completa, com informações de todos os participantes do sistema.

O HyperDHT foi apresentado como uma DHT de salto único que utiliza o algoritmo DiVHA para a construção e manutenção da rede de sobreposição, herdando os procedimentos de diagnóstico distribuído, bem como o limite máximo para latência. Mostrou-se que a garantia da latência máxima proveniente do sistema de diagnóstico, somado ao fato de todos os nodos terem conhecimento completo do sistema, permitem ao HyperDHT adotar estratégias onde as consultas realizadas são respondidas em um único salto, mesmo nas situações em que o sistema não está estabilizado com eventos ainda em fase de disseminação.

Um dos pontos de destaque do HyperDHT, em comparação com outras DHTs de salto único, é a incorporação de mecanismos no protocolo de entrada, para posicionar deterministicamente um novo participante no local da rede onde ele é mais necessário. Em geral, as demais DHTs deixam o problema do posicionamento de um novo participante por conta da função *hash*, acreditando que o efeito avalanche irá fazer uma distribuição razoavelmente homogênea dos participantes na rede de sobreposição.

Este trabalho apresenta os protocolos e algoritmos necessários para construção de

uma DHT de salto único utilizando o DiVHA. Entre os procedimentos estão: o particionamento, o mapeamento consistente, o balanceamento das chaves *hash*, a especificação dos protocolos de entrada e saída de participantes e, os mecanismos para posicionamento e busca dos pares chave/valor. Além disso, também são especificados mecanismos para replicação dos pares chave/valor a fim de evitar a falha de consultas.

Foram conduzidas duas séries de simulações. A primeira série teve como objetivo observar a latência requerida em diversos tamanhos de sistemas e, a segunda visou mostrar a influência de diferentes cargas de *churn* na ocorrência de *gets* em múltiplos-saltos. Em ambas as séries também foram coletados dados para análise do consumo dos recursos de rede.

A principal contribuição deste trabalho é a proposta de um novo modelo de *Single Hop DHT* escalável, auto-reorganizável, puramente P2P, de baixa latência e de baixo custo de manutenção. Outros aspectos do projeto também podem ser considerados contribuições, como a utilização de uma rede de sobreposição baseada em um hipercubo, a inserção de novos *peers* realizada de maneira determinística visando balanceamento de carga, e os procedimentos que aumentam a disponibilidade dos valores em consultas de um único salto.

Os trabalhos futuros serão iniciados portando-se o código implementado no simulador para a execução de experimentos em um ambiente mais próximo do modelo real, como o *PlanetLab*. Desta forma, também será possível a realização de experimentos com uma quantidade de *peers* superior a 512, pois os processos relacionados a cada *peer* estarão distribuídos por diversas máquinas, não causando sobrecarga de processamento em uma única máquina, como ocorreu durante a simulação.

Outro ponto que poderá ser considerado em trabalhos futuros, é a análise dos efeitos que *peers* maliciosos podem causar na rede HyperDHT, e as possíveis contra-medidas que podem ser tomadas para proteger o sistema durante este tipo de ataque.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] B. Liskov A. Gupta e R. Rodrigues. Efficient routing for peer-to-peer overlays. *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, Março de 2004.

[2] D. Androutsellis-Theotokis, S.; Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 2004.

[3] Jr.; Buskens R. Bianchini, R. An adaptive distributed system-level diagnosis algorithm and its implementation. *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, 1991.

[4] Kenneth P. Birman. Reliable distributed systems: Technologies, web services, and applications. *Springer*, 2005.

[5] M. Naor D. Malkhi e D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. *In Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, Julho de 2002.

[6] D. Doval, D.; O'Mahony. Overlay networks: a scalable alternative for p2p. *IEEE Internet Computing*, Agosto de 2003.

[7] A.; Albini L. C. P. Duarte Jr., E. P.; Brawerman. A diagnosis algorithm based on clusters with detours. 1998.

[8] Xiuqi Lee e Jie Wu. Searching techniques in peer-to-peer networks. *Handbook of Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks - Auerbach Publications, Boca Raton*, 2006.

[9] Alessandro Brawerman Andre L. P. Guedes Elias P. Duarte Jr., Luiz C. P. Albini. A hierarchical distributed fault diagnosis algorithm based on clusters with detours.

The 6th IEEE Latin American Network Operations and Management Symposium (LANOMS), 2009.

[10] Luiz C. P. Albini Elias P. Duarte Jr., Roverli P. Ziwich. A survey of comparison-based system-level diagnosis. *ACM Computing Surveys, ISSN 0360-0300*, 2011.

[11] P. Fraigniaud e P. Gauron. The content-addressable network d2b. *Technical Report LRI-1349, Universie de Paris Sud*, Janeiro de 2003.

[12] D. Karger R. Morris H. Balakrishnan, F. Kaashoek e I. Stoica. Looking up data in p2p systems. *Communications of ACM*, 2003.

[13] S.-Y. Hsieh e Y.-S. Chen. Strongly diagnosable systems under the comparison model. *IEEE Transactions on Computers*, 2008.

[14] Amiya Nayak Xiaofan Yang Hui Yang, Mourad Elhadef. An evolutionary approach to system-level fault diagnosis. *IEEE Congress on Evolutionary Computation*, 2009.

[15] D. Karger M. Kaashoek H. Balakrishnan I. Stoica, R. Morris. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of SIGCOMM'01, San Diego, CA, USA*, Agosto de 2001.

[16] Tim Moors John Risson, Aaron Harwood. Stable high-capacity one-hop distributed hash tables. *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC'06)*, 2006.

[17] E. P. Duarte Jr. e T. Nanya. A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Transactions on Computers*, 1998.

[18] M. Kaashoek e D. Karger. Koorde: A simple degree-optimal distributed hash table. *In Proceedings of IPTPS*, Fevereiro de 2003.

[19] E.; Leighton T.; Panigrahy R.; Levine M.; Lewin D. Karger, D.; Lehman. Proceedings of the twenty-ninth annual acm symposium on theory of computing. *ACM Press New York, NY, USA*, 1997.

[20] J. G. Kuhl e S. M. Reddy. Distributed fault tolerance for large multiprocessor systems. *In Proc. IEEE Sym.p. Comput. Architecture*, 1980.

[21] J.; Pias M.; Sharma R.; Lim S. Lua, E. K.; Crowcroft. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 2005.

[22] Keiko V. O. Fonseca Samuel L. V. Mello Luis C. E. Bona, Elias P. Duarte Jr. Hyperbone: A scalable overlay network based on a virtual hypercube. *The 8th International Symposium on Cluster Computing and the Grid (CCGRID 2008)*, 2008.

[23] Claudio L. Amorim Luiz R. Monnerat. D1ht: A distributed one hop hash table. *Technical Report ES-705/06, COPPE/UFRJ*, 2006.

[24] M.H. MacDougall. Simulating computer systems: Techniques and tools. *The MIT Press, Cambridge, MA*, 1987.

[25] Adriana Iamnitchi Matei Ripeanu e Ian T. Foster. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, Janeiro de 2002.

[26] P. Maymounkov e D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *In International Workshop on Peer-to-Peer Systems*, 2002.

[27] F. Schreiner D. Vingarzan N. Blum, P. Jacak e P. Weik. Towards standardized and automated fault management and service provisioning for ngns. *Journal of Network and Systems Management*, 2008.

[28] National Institute of Standards e Technology. Fips 180: Secure hash standard. *Disponível em: http://csrc.nist.gov*, Maio de 1993.

[29] A. Gupta P. Fonseca, R. Rodrigues e B. Liskov. Full information lookups for peer-to-peer overlays. *IEEE Transactions on Parallel and Distributed Systems*, Setembro de 2009.

[30] Rajaraman R. Plaxton, C. e A. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Proceedings of ACM SPAA. ACM*, 1997.

[31] Handley M. Karp R. Ratnasamy S., Francis P. e Shenker S. A scalable content-addressable network. *ACM SIGCOMM, San Diego, CA, USA*, Agosto de 2001.

[32] Geels D. Roscoe T. Rhea, S. e J. Kubiatowicz. Handling churn in a dht. *In Proceedings of the USENIX Technical Conference*, Junho de 2004.

[33] A. Rowstron e P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*, Novembro de 2001.

[34] Ronald L. Rivest e Charles E. Leiserson Thomas H. Cormen, Clifford Stein. Introduction to algorithms. *McGraw-Hill Higher Education*, 2001.

[35] J. Xu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. *Proceedings of the IEEE Conference on Computer Communications - INFOCOM*, Maio de 2003.

[36] X. Yang e Y. Y. Tang. Efficient fault identification of diagnosable systems under the comparison model. *IEEE Transactions on Computers*, 2007.

[37] Kubiatowicz J. Zhao, B. Y. e A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. *Tech. Rep., University of California at Berkeley, Computer Science Department*, 2001.

JEFFERSON PAULO KOPPE

HYPERDHT - DHT DE UM SALTO BASEADA EM HIPERCUBO VIRTUAL DISTRIBUÍDO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Luis Carlos E. de Bona
Co-Orientador: Prof. Dr. Elias P. Duarte Jr.

CURITIBA

2013